UNIVERSITÉ DE TUNIS
INSTITUT SUPÉRIEUR DE GESTION

MÉMOIRE DE MASTER RECHERCHE

SPÉCIALITÉ

Sciences et Techniques de l'Informatique Décisionnelle

OPTION

Informatique et Gestion de la Connaissance (IGC)

Evaluation of RDF Archiving strategies with Spark

MERIEM LAAJIMI

SOUS LA DIRECTION DE:

NADIA YAACOUBI MAITRE ASSISTANT, ISG TUNIS
AFEF BAHRI MAITRE ASSISTANT, ESC TUNIS

LABORATOIRE RIADI

ANNÉE UNIVERSITAIRE 2016 / 2017

Acknowledgments

First and foremost, I would like to express my deepest gratitude to Allah for giving me the strength, patience and wisdom to complete this work.

I would like to express my special thanks of gratitude to my supervisors Dr Afef Bahri and Dr Nadia Yaacoubi for their support, suggestions, guidance, care, motivations and great effort they made throughout my study.

I would also like to thank all my family and my friends for their love, encouragement and continuous support .

Finally, I would like to express my sincere gratitude to all of my professors of the Higher Institute of management of Tunis.

Contents

List of Figures

List of Tables

Introduction

The Linked Data paradigm promotes the use of the RDF model to publish structured data on the Web. As a result, several datasets have emerged incorporating a huge number of RDF triples. Some datasets are cross-domain such as Dbpedia and Freebase, others are dedicated to a specific domain (geography, Life science, media, government, etc.) such as Bio2RDF[1]. The Linked Open Data cloud[2], as published in 22 August 2017 illustrates the important number of published datasets and their possible interconnections. On another side, LODstats a project constantly monitoring statistics reports 2,973 RDF datasets that incorporate approximately 149 billion triples. Note that these RDF datasets are automatically populated by extracting information from different resources (Web pages, databases, text documents) leading to an unprecedented volume of RDF triples. Indeed, published data is continuously evolving and it will be interesting to manage not only a current version of a dataset but also previous ones. In fact, users would like to query previous versions, compare different versions or to lookout the evolution of a specific data among different versions. There is an emerging interest on what we call archiving of Linked Open Data (Stefanidis, Chrysakis, & Flouris, 2014; Fernández, Umbrich, Polleres, & Knuth, 2016; Meimaris & Papastefanatos, 2016) and several challenges need to be addressed.
The absence of a central control makes impossible the propagation and the tracking of changes to all the related parts. That is, semantic applications need to access to the previous versions in order to query and track data over time. Thus, dealing with cross-version or time-traversing queries becomes an important challenge for

[1] http://bio2rdf.org/

[2] Linking Open Data cloud diagram 2017, by Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch and Richard Cyganiak. Available at http://lod-cloud.net/

archiving systems. Theses queries are very useful to identify the state of a dataset at a given time, to compare different versions of a dataset and to analyze and monitor the evolution process.

Three versionning strategies are adopted in RDF archiving systems and cited in literature as follows: (a) Independent Copies (IC), (b) Change Based copies (CB) or Deltas and (c) Timestamp-based approaches (TB) (Papakonstantinou, Flouris, Fundulaki, Stefanidis, & Roussakis, 2016). Initial works on RDF dataset archiving are focusing on the two first approaches (Stefanidis et al., 2014; Fernández et al., 2016). The first one is a naive approach since it manages each version of a dataset as an isolated one. Obviously, scalability problem is expected due to the large size of duplicated data across dataset versions. The delta-based approach aims to resolve (partially) the scalability problem by computing and storing the differences between versions. While the use of deltas reduces space storage, the computation of full version on-the-fly may cause overhead at query time. Therefore, archiving systems not only need to store and provide access to different versions, but should also be able to support various types of queries on the data that a user may need to formulate (Fernández, Umbrich, & Polleres, 2015; Stefanidis et al., 2014). In an archiving system, queries may serve different needs and focus ranging from version or delta materialization to single-version or cross-version queries also called time-traversal queries. The latter type of queries are the most significant and complex one in terms of processing since queries are evaluated across dataset versions. In this context, the independent copies strategy may allow an efficient response time for certain kind of queries (version materialization, single version query) but needs to face scalability problems. To resolve this issue, authors in (Stefanidis et al., 2014) propose hybrid archiving policies to take advantage of the independent copies archiving approach and the delta one. In fact, a cost model is conceived to determine what to materialize at a given time: a version or a delta.

Moreover, the emergent need for efficient web data archiving leads to recently developed Benchmarking RDF archiving systems such as BEAR (BEnchmark of RDF ARchives) (Fernández et al., 2015) and EvoGen (Meimaris & Papastefanatos, 2016). The authors of the BEAR system propose a theoretical formalization of an RDF archive and conceive a benchmark focusing on a set of general and abstract queries with respect to the different categories of queries as defined before. More recently, the EU H2020 HOBBIT[3] project is focusing the problem of Benchmarking Big Linked Data. A new Benchmark SPBv was developed

[3]https://project-hobbit.eu/

with some preliminary experimental results (Papakonstantinou, Flouris, Fundulaki, Stefanidis, & Roussakis, 2017). Similar to EvoGen, SPBv proposes a configurable and adaptive data and query load generator.
Obviously, the fast increasing size of RDF datasets raises the need to treat the problem of RDF archiving as a Big data problem. Many efforts has been done to process RDF linked data with existing Big data processing infrastructure like Hadoop or Spark (Naacke, Curé, & Amann, 2016; Schätzle, Przyjaciel-Zablocki, Skilevic, & Lausen, 2016). Nevertheless, no works has been realized for managing RDF archives on top of cluster computing engine. The problem is more challenging here as Big data processing framework are not designed for RDF processing nor for evolution management.

Aim and Scope

In this master thesis, we use the in-memory cluster computing framework SPARK for managing and querying RDF data archives using independent copies and change based approaches. We give a formal modeling of RDF dataset archives and change operations. We propose a theoretical formalization of RDF versioning queries equally defined using SPARK SQL syntax. Mapping rules from SPARQL to SPARK SQL are proposed. We note that, different SPARQL query shapes may produce a certain number of join operations between triple patterns. We show in this master thesis how the use of version based query may increase this problem. We propose an evaluation of main versioning queries on top of SPARK framework using Scala. Different performance tests have been realized based on: versioning approaches (Change Based or Independent Copy approaches), the types of RDF archives queries, the number of versions, the shape of SPARQL queries and finally the data partitioning strategy.

Master Thesis Outline

This master thesis is organized into two main parts.
The first part, Theoretical aspects, is composed of two main chapter which are the following:

- Chapter 1 presents basic concepts related to Linked Data, RDF and SPARQL languages, then, we present the Big data concept and some key technologies developed in this context.
- Chapter 2 presents existing approaches for the design and evaluation of RDF archiving and versioning systems.

The second part of this master thesis presents our contributions and is composed of two chapters:

- Chapter 3 presents our approach for RDF datasets archives modeling and we provide a theoretical formalization of RDF versioning queries. SPARK is used to query RDF dataset archives and mapping of SPARQL to SPARK SQL is proposed.
- Chapter 4 presents an evaluation of RDF versioning queries on top of SPARK.

Finally, the conclusion summarizes all the work presented in this report and proposes a future works.

Part I
Theoretical Aspects

Part I presents the theoretical aspects of this master's thesis. Chapter 1 introduces the basic concepts of linked data and RDF. Then, it details the principles of big data including its challenges and techniques. Chapter 2 gives an overview about the RDF archiving systems including its storage strategies and querying.

Chapter 1

Preliminaries

1.1 Introduction

In this chapter, we give an overview of the basic concepts and background terminology of our research work. Section 1.2 presents the Linked data paradigm. Section 1.3 gives an overview of the RDF data model as well and the SPARQL query language as the principles techniques used in the context of Linked data. Section 1.4 presents the principles and challenges of Big data and section 1.5 introduces some of the techniques developed in this context.

1.2 Linked data

The Linked Data paradigm was firstly introduced by Tim-Berners Lee in 2006 (Bizer, Heath, & Berners-Lee, 2009). It refers to a set of guidelines and best practices for publishing and interlinking structured data on the semantic web using standard technologies such RDF and URI. The Linked Data may be seen as a global data space containing thousands of datasets resulting from diverse domains such as people, companies, books, scientific publications, films, music, television and radio programs, genes, proteins, drugs and clinical trials, online communities, statistical and scientific data, and reviews. These datasets are illustrated by the different colors in the Linked Open Data cloud depicted in figure 1.1[1].

[1] http://lod-cloud.net

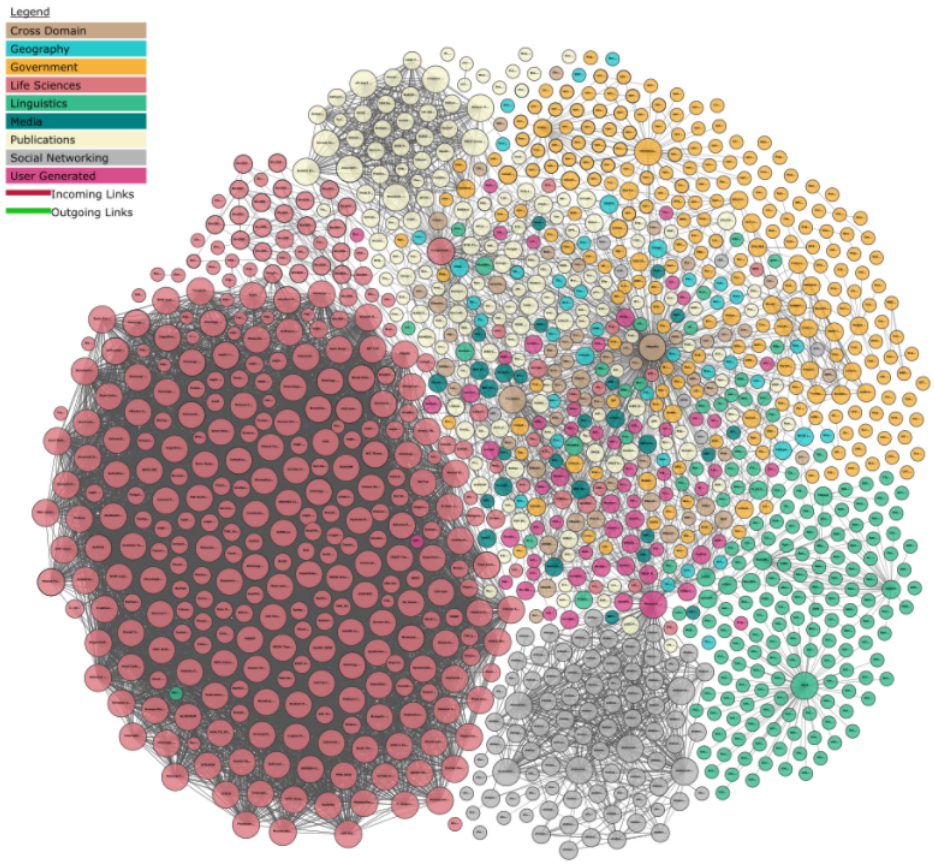

Figure 1.1: Linked Open Data Cloud

Berners-Lee summarizes the set of principles for publishing data on the Web (Bizer et al., 2009):

- Use URI to name things.
- Use HTTP URIs so people can be looked up.
- When someone looks up for a URI, provide useful information via RDF (and SPARQL).
- Include links to other URIs, so that they can discover more things.

Over the last few years, the number of datasets published in Linked Data has increased from 12 RDF datasets in 2007, to more than 2 thousands RDF datasets in 2017 and this number is continously increasing.

1.3 Representation and Querying of Semantic Data

In this section we respectively present RDF and SPARQL query language and we define the concept of RDF triple Store.

1.3.1 RDF Data Model

The Resource Description Framework (RDF) is a graph data model proposed by the W3C as a standard for representing information about resources in the Web (Huang, Abadi, & Ren, 2011). It provides a flexible mechanism to define metadata describing real world objects. The basic construct in RDF is the triple (s,p,o) called also RDF statement which is composed of the following three parts:

- Subject: It identifies the object of the triple that is being described.
- Predicate: It describes the relationship between subject and object.
- Object: It represents the value of the RDF triple.

A set of triples is also called an RDF graph which is used as a basis for expressing information across different fields and domains. The different entities in the graph are represented as vertex and the relationship between them as edges. The subject and the object are denoted as vertices, while the predicate is represented as labeled edge. We distinct three kinds of nodes in an RDF graph:

- URI node: node that corresponds to an URI (Uniform Resource Identifiers) and is used to uniquely identify a resource.
- Blank node: does not have an identifier and it can not be referenced from outside.
- Literal node: is used to represent property values such as texts, numbers and dates.

In an RDF triple t = (s,p,o), the subject s is either an URI or a blank node, the predicate p is an URI and the object o is either an URI, a blank node or a literal. Thus, an RDF triple is formally defined as follows.

Definition 1.1 (RDF triple) Given disjoint finite sets U, B and L denoting respectively sets of URIs, blank nodes and Literals, a triple (s, p, o), $t \in (I \cup B) \times I \times I \cup B \cup L$ is s called RDF triple where s, p and o represent respectively the subject, predicate and object of the triple.

Figure 1.2 gives an example of an RDF triple where the subject identified with URI http://www.disco.unimib.it/go/45827 is "author" of the paper identified with URI http://ceur-ws.org/Vol-1605/paper3.pdf.

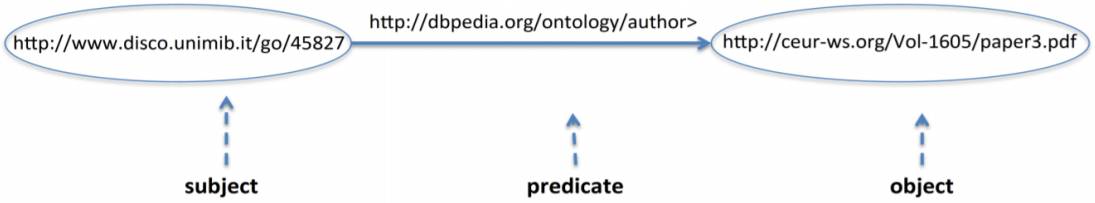

Figure 1.2: RDF triple example (Spahiu, 2017)

Figure 1.3 illustrates an RDF graph. The vertices in the graph indicate that the entity identified by the URI "http://www.disco.unimib.it/go/45827" has type 'person', her name is 'Belerina Spahiu' and her date of birth is '9/10/1986'.

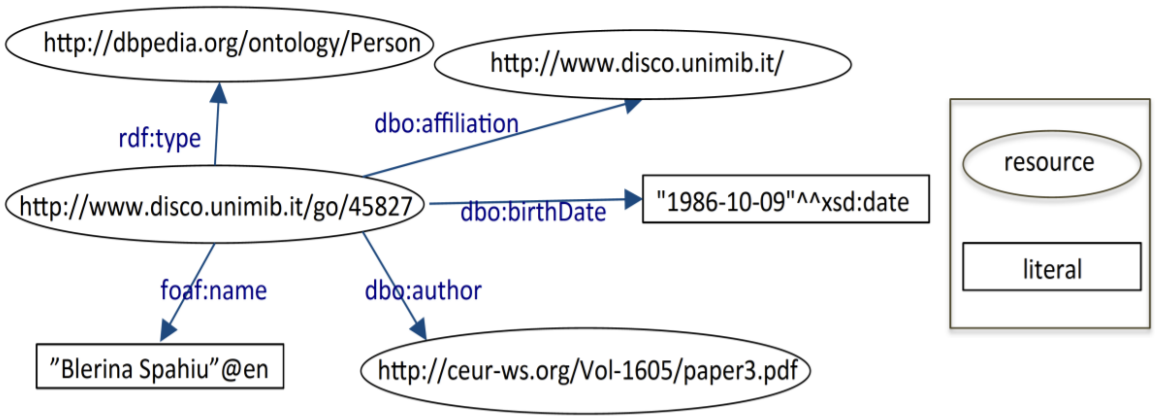

Figure 1.3: RDF Graph Example (Spahiu, 2017)

1.3.2 SPARQL

SPARQL (Recursive acronym: SPARQL Protocol And RDF Query Language) is the W3C (WWW Data Access Working Group) recommended language for querying RDF data (Spahiu, 2017). A SPARQL query contains a set of triple patterns called a basic graph pattern. SPARQL provides a set of analytic query operations such as JOIN, SORT and AGGREGATE. We distinct four different query variations:

- **SELECT query**: used to extract raw values from a SPARQL endpoint (a service that accept queries and return result).
- **CONSTRUCT query**: used to extract information from the SPARQL endpoint and transform the results into valid RDF.
- **ASK query**: used to provide a simple True/False result for a query on a SPARQL endpoint
- **DESCRIBE query**: used to extract an RDF graph from the SPARQL endpoint. The content of the query result is left to the endpoint to decide based on what the maintainer deems as useful information.

```
01.  PREFIX dbo: <http://dbpedia.org/ontology/>
02.  PREFIX res: <http://dbpedia.org/resource/>
03.  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
04.
05.  SELECT DISTINCT ?uri
06.  WHERE {
07.       ?uri rdf:type dbo:Book .
08.       ?uri dbo:author res:Haruki_Murakami .
09.  }
```

Figure 1.4: SPARQL Query Example (Spahiu, 2017)

Figure 1.4 presents an example of a SPARQL query. The query begins with PREFIX statements defining the abbreviation for the namespace (lines 01 to 03). SELECT clause (line 05) contains a variable named "?uri". The WHERE clause is defined in line 06. Line 07 specifies that we need those resources that are of

type Book, while line 08 states that we are looking for those resources for which the author is "Haruki Murakami". The result of the SPARQL query is shown in figure 1.5.

uri
http://dbpedia.org/resource/A_Wild_Sheep_Chase
http://dbpedia.org/resource/Sputnik_Sweetheart
http://dbpedia.org/resource/After_the_quake
http://dbpedia.org/resource/Blind_Willow,_Sleeping_Woman
http://dbpedia.org/resource/Hard-Boiled_Wonderland_and_the_End_of_the_World
http://dbpedia.org/resource/Kafka_on_the_Shore
http://dbpedia.org/resource/Norwegian_Wood_(novel)
http://dbpedia.org/resource/South_of_the_Border,_West_of_the_Sun
http://dbpedia.org/resource/The_Wind-Up_Bird_Chronicle
http://dbpedia.org/resource/Underground_(Murakami_book)
http://dbpedia.org/resource/Dance_Dance_Dance_(novel)
http://dbpedia.org/resource/Pinball,_1973
http://dbpedia.org/resource/Uten_Enten
http://dbpedia.org/resource/Hear_the_Wind_Sing
http://dbpedia.org/resource/What_I_Talk_About_When_I_Talk_About_Running
http://dbpedia.org/resource/After_Dark_(Murakami_novel)
http://dbpedia.org/resource/Colorless_Tsukuru_Tazaki_and_His_Years_of_Pilgrimage
http://dbpedia.org/resource/1Q84
http://dbpedia.org/resource/The_Elephant_Vanishes

Figure 1.5: An SPARQL query result example (Spahiu, 2017)

1.3.3 RDF Triple store

Triple store are databases designed for RDF data storage. The RDF datasets are indexed in Triple store to facilitate the execution of SPARQL queries. The most knows triple stores are Virtuoso (Wauer, Both, Schwinger, Nettling, & Erling, 2015) and Sesame (Broekstra, Kampman, & van Harmelen, 2002). On the other side, many systems store the RDF data in relational database system (DBMS).

Each RDF statement is stored as a triple in one large table with a three-columns schema (i.e. a column for subject, predicate and object).

1.4 From Big data to Linked data

1.4.1 Big data paradigm

For a long time, the volume and the variety of data have outstripped the capacity of manual analysis and exceeded the capacity of conventional databases in some case. The term Big Data can be defined as a large amount and complex datasets that the traditional technologies can not process. Several dimensions is defined in conjunction with big data, namely the 5V concept which denote: Volume, Variety, Velocity Veracity and Value (Anuradha et al., 2015). The first dimension **Volume** refers to the quantity of data collected and captured from different sources such as: sensors, mobile device, social network. The second dimension **Variety** is about the type of data that can be classified in different categories from structured, unstructured, standard, semi-structured and raw data which are very difficult to be handled by traditional systems. The third dimension **Velocity** refers to the speed of generated data from different sources. The fourth dimension **Veracity**, the main goal of data analysis is to extract useful information from high volume of data in order to have good results. The last dimension **Value** is the most important aspect in big data, we can have access to massive data but unless we can turn into value it is become useless.
Obviously, the big data is omnipresent in many fields and sectors such as Business, Finance, Medicine, Bioinformatics, social networks, etc. Therefore, big data management involves several tasks such extraction, storing, analysis and visualization. Each task introduces critical challenges which can be summarized principally into data inconsistency, scalability and timeliness. Datasets are collected from different sources which may lead to data inconsistency, thus, pre-processing treatments are necessary in order to improve data quality. The scalability issue deals with the efficiency of processing capabilities to deal with continuously growing amounts of data but the big challenge of big data is to guarantee response's timeliness when processing data streams. (Chen & Zhang, 2014). Several technologies of Big data was conceived to deal with these challenges. An overview of theses technologies is given in section 1.5.

1.4.2 Scaling up Linked data

Given that the web is growing rapidly and submerged by a huge amount of datasets, we can associate it the notion of big data. Recently, the big data has been drawn the attention of many researchers. Big Data and Linked Data can be seen to complement in each other in two important ways. First, big data techniques can be applied to Linked Data challenges. As described above, there has been a huge growth in Linked Data over the past five years. Techniques developed in the Big Data research community can be adopted to handle large amounts of Linked Data. Second, Linked Data techniques can be applied to Big Data challenges. Big Data can have a high level of variety and Linked Data techniques can be applied to the problem. Linked Data tools can be used to enrich legacy content and improve data discovery and integration. Interlinking datasets using a common format can help to reduce data duplication.

1.5 Big Data Tools

Many tools have been proposed for processing massive data efficiently and managing data distribution among multiple machines. Apache Hadoop and Apache Spark are the most popular open source frameworks.

1.5.1 Hadoop framework: HDFS and MapReduce

Apache Hadoop is an open source framework which supports distributed process and distributed storage (Anuradha et al., 2015). Two main subsystems compose Apache Hadoop: the Hadoop Distributed File System (HDFS) for data storage and the Map Reduce model for data processing.

HDFS

HDFS is a distributed and scalable file system (Anuradha et al., 2015). It stores a large amount of data (typically in the range of gigabytes to terabytes) in reliable manner by providing a highly fault tolerant file system. The reliability is achieving by replicating data accros multiple nodes even if some machine crashes. The

HDFS architecture shown in figure 1.6[2] is composed of two kinds of nodes. A name node (master) and a number of data nodes (workers). The master node takes the charge of managing the name space by saving directories, opening and closing files. It also stores the metadata which contains information like the number of replication of blocks and the location of each block. In case of failure of name node, the system provides the secondary name node connected with the primary one. The workers nodes are used for storing and retrieving data blocks and ensure several operations such as create, delete and replicate.

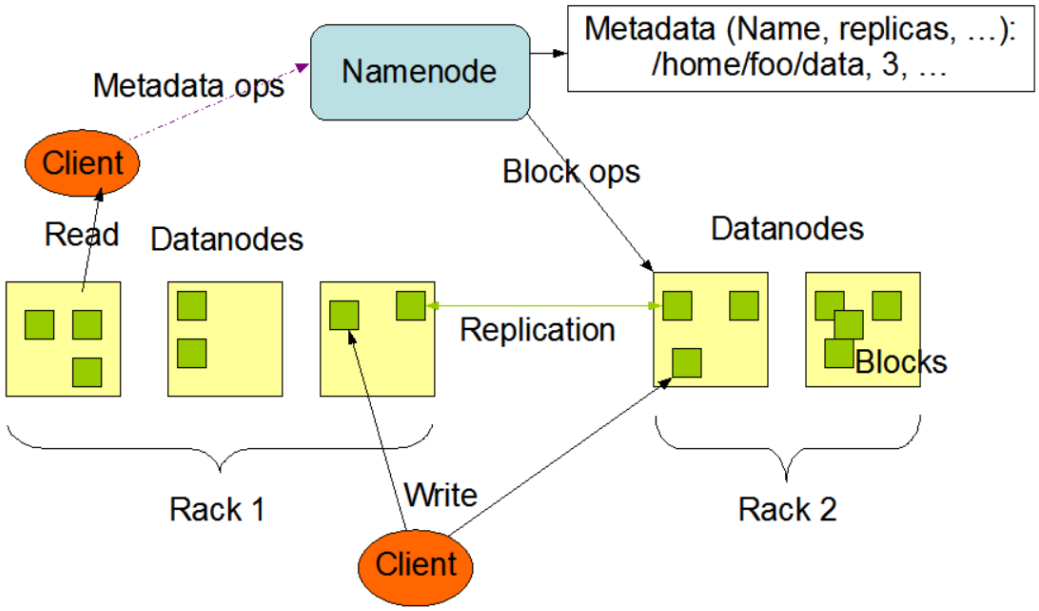

Figure 1.6: HDFS Architecture

MapReduce

MapReduce is a programming model invented by Google and developed by yahoo, it has been widely used to parallelize the computation across a cluster of machines (Dean & Ghemawat, 2008). It provides a parallel design pattern for simplifying application developments in distributed environments. Obviously, every MapReduce execution requires the master nodes and the workers. Indeed, the master node distributes the task to the worker nodes. Concerning this phase, tasks

[2] `https://hortonworks.com/apache/hdfs/`

can execute two different functions: Map or Reduce (Memishi, Ibrahim, Pérez, & Antoniu, 2016). A MapReduce workflow scenario, depicted in figure 1.7, is defined as below:

- **A map phase**: The map function takes a key/value pairs as input and produces a list of key/value pairs as output and intermediate results.
- **A shuffle phase** :All the intermediate results are grouped by keys.
- **A reduce phase** : The reduce function invokes once for each key with associated values and generates a list of output values as final results.

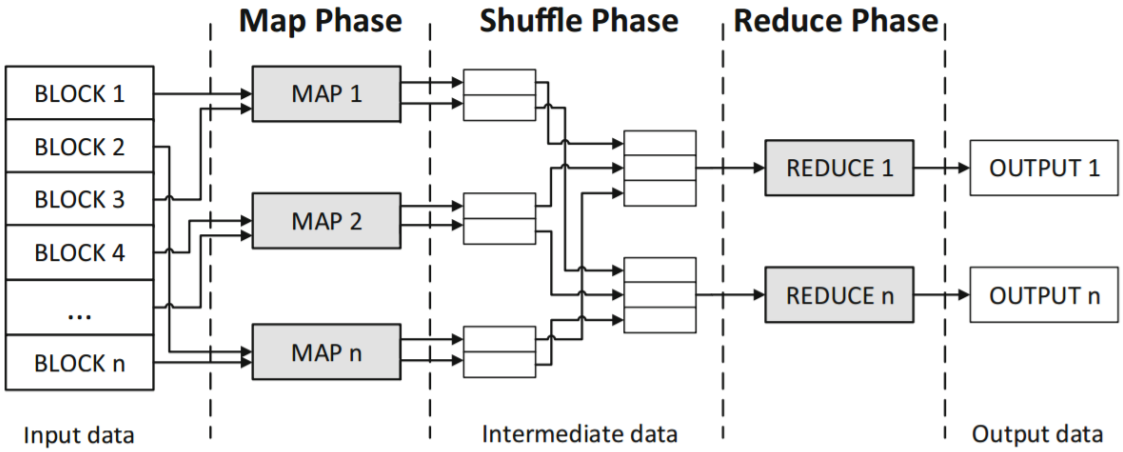

Figure 1.7: MapReduce workflow (Memishi et al., 2016)

The MapReduce paradigm has many advantages. We can cite among others:

- Scalable: because it stores and distributes large data sets on plenty of machines which operate in parallel.
- Flexible: since it can access to various new sources of data and also operates on different types of data whether they are structured or unstructured.
- Fault tolerant: is ensured by the replication of the datasets on different machines in case of machine failure.

Apache Hadoop system contains a wide range of big data tools(Anuradha et al., 2015) in order to facilitate the storage and make easy the analyzing and querying of data. We present the most used ones:

- **Hive**: is a data warehouse that used the HiveQL language to analyze the data. HiveQL seems like standard SQL but it provides faster and secure access to the distributed data set.
- **Hbase**: is a distributed, scalable, an open NoSQL database implemented on top of HDFS. It offers a real-time access to huge volume of multi-structured data set.
- **Pig**: is a platform for analyzing large datasets that consists of a high-level language 'the Pig Latin language' to perform complex operations.

1.5.2 Apache Spark

Spark is an open source framework build to perform sophisticated analysis(Zaharia, Chowdhury, Franklin, Shenker, & Stoica, 2010). It was developed by AMPLab at the University of California, Berkeley in 2009 then the codes were donated to the Apache Software Foundation in 2013 to become one of the Foundation's top-level projects in 2014. It is based on the concept of saving data in memory rather than a disk and exploits in-memory computation for solving iterative algorithms. It also supports multiple languages like Java, Scala, or Python and provides a powerful and user friendly API (Application Programming Interface) for a better productivity.
The feature that makes Apache spark distinctive compared to Hadoop is the new distributed memory abstraction called Resilient Distributed Data sets. The resilient distributed data (RDD) is a set of objects immutable and distributed across a cluster machines that form the main core of Apache Spark. The RDDs allow to rearrange the calculations as a goal to optimize the treatment. They are fault tolerant since they can be recreated and recalculated if a partition is lost (Zaharia et al., 2016a).
Two types of operations can be performed in an RDD: produce other RDDs (transformations) or return values (actions):

- **Transformation**: This function apply such changement on RDD to returns a new RDD. The transformation functions are for example map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, pipe and coalesce.
- **Actions**: This function evaluates and returns a new value. Actions include reduce, collect, count, first, take, countByKey, and foreach.

Spark ecosystem

Apache spark is an unified parallel computation project that contains multiple components to support efficiently many complex applications. Figure 1.8 [3] depicts the Apache Spark ecosystem with its components (Meng et al., 2016):

- **Spark core**: It represents the most important component and it contains the basic spark functionalities such as task scheduling, managing memory and creating RDDs.
- **Spark Streaming**: This module is used for processing real-time data stream. A series of RDDs compose theses data streams which provide the same operations as typical RDDs enriched with new ones.
- **Spark SQL**: This module allows users to handle a variety of data sources using SQL. Spark SQL extract, transform and load data in different formats (JSON, Parquet, database).
- **MLlib**: MLlib is a machine learning library that contains many machine learning algorithms such as K-means, SVM and decision trees.
- **Spark GraphX**: GraphX was designed to perform parallel computation of graph's algorithm such as PageRank and shortest path, etc.

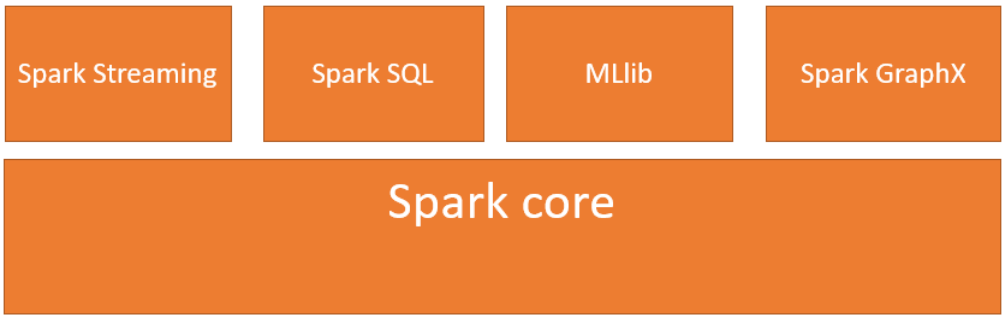

Figure 1.8: Apache Spark ecosystem

Spark provides properties like scalability and fault tolerance similar to MapReduce and allows the development of pipelines for processing complex data using oriented acyclic graphs (DAG). It allows to share data in memory between graphs so that multiple jobs can work on the same dataset.

[3] `https://databricks.com/spark/about`

Spark architecture

Apache spark loads at first the input data from an external data storage such as HDFS, and Amazon S3 in order to create RDDs. A spark cluster is composed essentially by the driver node and a set of spark workers. The driver instantiates an object SparkContext to establish the connection to the cluster and hold in the users driver program. Nevertheless, the cluster manager controls the resources allocation, manages the errors and fixes the task scheduling to launch them to the worker nodes. The workers execute the tasks as assigned by the driver and stored data locally. The overall flow is described in figure 1.9[4].

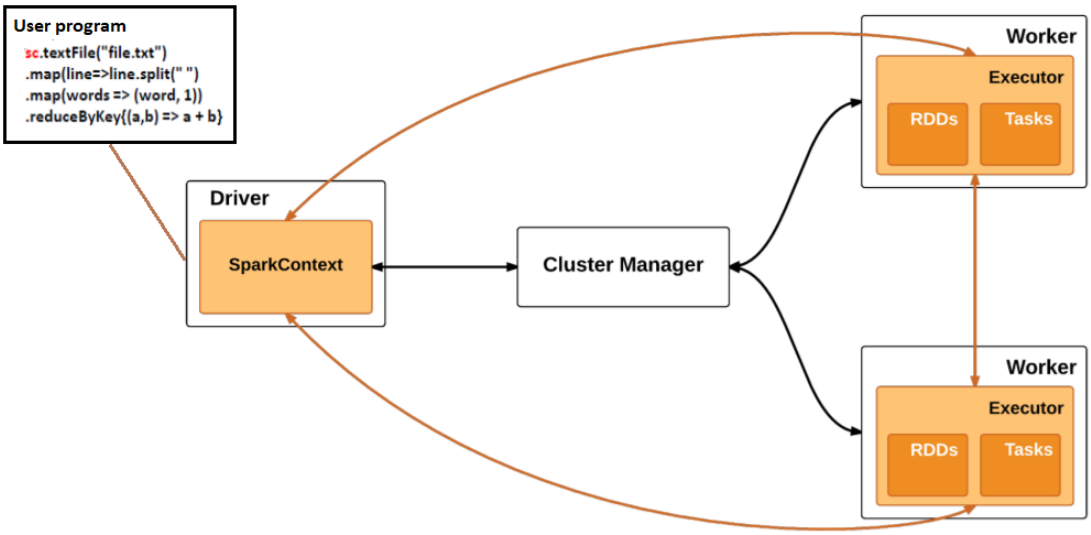

Figure 1.9: Spark architecture

1.5.3 MapReduce Vs Spark

The MapReduce and Spark share obviously the same advantages by hiding the complexity of task parallelism and allowing fault-tolerance. MapReduce seems to be a good solution for processing a large volumes of data but it has several limitations. In cluster computing, each map and reduce phase save operations in the disk which engenders extra delay in execution. Equally, the replication of data takes a lot of space in disk storage.

[4]https://www.analyticsvidhya.com/blog/2016/09/comprehensive-introduction-to-apache-spark-rdds-dataframes-using-pyspark/

In order to remedy this limitations, Spark brings improvements to MapReduce through the concept of RDD that allows to store data in memory and this engender the increasing of the performance (ten hundred times more than the conventional MapReduce) and allowing more flexibility. Apache spark is judged to be the most suitable one because it uses the lineage concept to ensure fault-tolerance while MapReduce requires to replicate data items among machines.

1.6 Conclusion

In this chapter, we defined the notion of Linked data, the RDF data model and the SPARQL query language. We also present the Big data paradigm and we focused on the way we may use it to scale up Linked data. We equally define some of the techniques used in the context of Big data. In the next chapter we focus on RDF archiving systems and we give a state-of-the-art of the different approaches proposed for representing and querying RDF dataset archives.

Chapter 2

RDF Archiving Systems: Storage Strategies and Querying

2.1 Introduction

Over the last decade, the published RDF data is continuously growing leading to the explosion of the Web of Data and the associated Linked Open Data (LOD) in various domains such as geographic information systems, people, companies, films, music, genes, drugs, books, and scientific publications. Typically, the open nature of the Web reflects the dynamicity of the Web of data where both the data and the schema of RDF LOD datasets are constantly evolving (e.g information enrichment, scientific knowledge is constantly growing). This evolution naturally happens without pre-defined policy nor a central control. Afterward, there is a need to build open data archiving systems having their own infrastructures in order to preserve and query RDF data over time.
In this chapter, section 2.2 presents a state of the art of different RDF storage strategies for versionning RDF archives. Section 2.3 defines principle versioning queries and their application on RDF datasets. Finally, section 2.4 presents the different Benchmarks proposed in the literature for RDF archiving and querying.

2.2 Storage strategies for versionning RDF archives

In the literature, three RDF versionning approaches have been proposed: Independent Copies (IC), Change-based (CB), and Time-stamp approaches (Zablith et al., 2015). We talk about hybrid approaches when the above techniques are combined (Sande et al., 2013; Stefanidis et al., 2014). We present in the following the principle of these approaches.

2.2.1 Independent Copies

Independent Copies is the basic approach where all the versions of an RDF dataset are stored independently. This approach is used for managing RDF ontologies (SemVersion (Völkel & Groza, 2006)), LOD datasets (Memento (de Sompel et al., 2010)) and RDF archives in EvoGen (Meimaris & Papastefanatos, 2016) and BEAR benchmarks (Fernández et al., 2016). A description of this approach is depicted in figure 2.1.

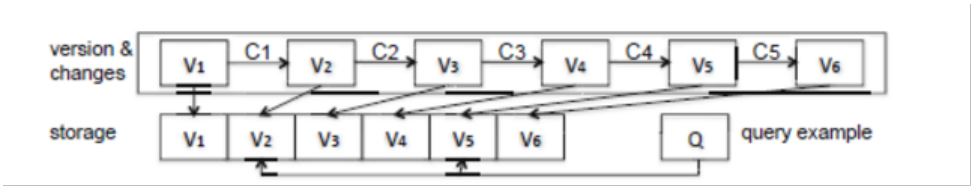

Figure 2.1: Independent Copies Approach (Stefanidis et al., 2014)

The advantage beyond the use of the IC approach is that simple retrieval operations such as version materialization is efficient as all the versions are already materialized in the archive. Nevertheless, the IC approach faced the scalability issue through the static information that is duplicated across versions. In the worst-case scenario, where we have a large versions changed frequently, the space overhead becomes huge.

2.2.2 Change-based approach

This approach treats the scalability issue by computing the differences between versions and storing only the changes that should be kept with respect to the previous version also known as the delta (Troullinou, Roussakis, Kondylakis, Ste-

fanidis, & Flouris, 2016). This approach is used for versioning RDF datastore in (Cassidy & Ballantine, 2007; IM, LEE, & KIM, 2012) and for RDF dataset archiving in EvoGen (Meimaris & Papastefanatos, 2016) and BEAR (Fernández et al., 2016) Benchmarks. The main asset of the change-based approach concerns the modest space requirements as deltas are much smaller than the dataset itself (figure 2.2). Nevertheless, the CB imposes additional computational costs for computing and storing deltas as many queries would require the on-the-fly reconstruction of one or more full versions of the data. Equally, this strategy needs an extra time overhead for delta propagation which involves the retrieving operations. For instance, as depicted in figure 2.2, in order to compute the difference between two non consecutive versions V_1 and V_5 we need to materialize all the versions that precede V_5 (V_2, V_3 and V_4) in order to retrieve the changes.

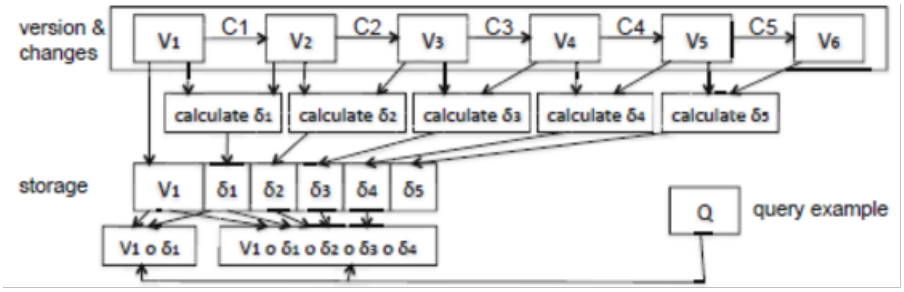

Figure 2.2: Change based approach (Stefanidis et al., 2014)

In order to improve the situation, different alternatives have been proposed such as computing reverse deltas by storing a materialization of the current versions and computing the changes with respect to this (Cassidy & Ballantine, 2007; IM et al., 2012; Graube, Hensel, & Urbas, 2014).

2.2.3 Time-stamp based approach

This approach introduces the notion of time modelling in RDF where each triple is annotated with its temporal validity (Gutierrez, Hurtado, & Vaisman, 2007; Tappolet & Bernstein, 2009; Neumann & Weikum, 2010; Udrea, Recupero, & Subrahmanian, 2010; Zimmermann, Lopes, Polleres, & Straccia, 2012). This annotation allows to return all triples that have been created before a given time t and were deleted after time point t and reconstructs the dataset version at any given time point. This approach permits to save space by avoiding such repetitions since the triples is annotated only when they are added or deleted. The BEAR

benchmark evaluates this approach on LOD RDF Archives in (Fernández et al., 2016). In figure 2.3, we observe that each triple holds the time-stamp of the version.

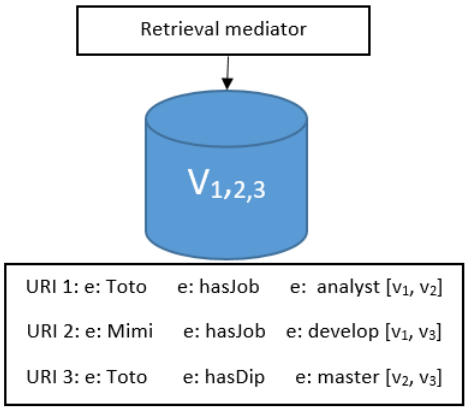

Figure 2.3: Time-stamp based approach

2.2.4 Hybrid approach

This approach combines the previous ones in order to take advantages of each approach. The combination of *Independent Copies* and *Change-Based* approaches is the most commonly used (Stefanidis et al., 2014). In fact, a cost-based model is used to choose which versions (not eventually all) are materialized while deltas are used to materialize the other versions if needed. The author of (Fernández et al., 2016) introduces this approach in the last version of the BEAR benchmark. As we can observe in figure 2.4, both versions (V_1, V_3 and V_6) and deltas (δ_1, δ_3, δ_4) are stored.

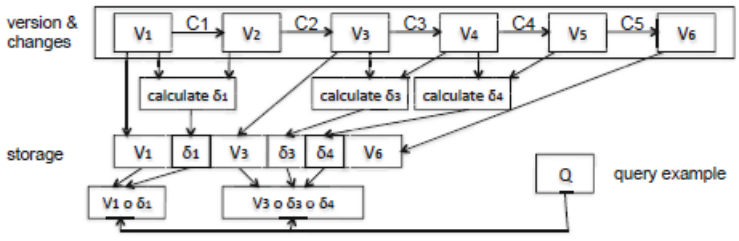

Figure 2.4: Hybrid approach (Stefanidis et al., 2014)

2.3 Query types

In this section, we enumerate the different types of queries that have been proposed in the literature. We can distinguish six different types of retrieval needs shown in Table 2.1. The classification that we adopted considers the query type (materialization or structered queries) and mainly focus (Version or Delta).

Type / Focus	Materialization	Single time	Cross time
Version	Version materialization: Get snapshot at time t_i.	Single-version structured queries: A query that asks for the number of friends that a certain person has at a given time.	Cross-version structured queries: A query that asks about all the status and updates of a specific person through time.
Delta	Delta materialization: Get delta at time t_i.	Single-delta structured queries: A query that asks for the new friends that a person obtained between two versions t_i and t_j.	Cross-delta structured queries : A query that asks about the changes occurred in the friends network of a person (e.g. friends added and/or deleted).

Table 2.1: Classification and examples of retrieval needs.

In the following we present successively Versions and Delta based queries.

2.3.1 Version query

Version query consider complete versions. We distinct *version materialization*, *single version structured queries* and *cross version structured queries*:

- **Version Materialization:** A basic demand in which we ask for a full current version to be retrieved. This can be seen as simple, indeed, this is the most

common feature provided by revision control systems and other large scale archives. This kind of retrieval functionality still presents a challenge as the size of triple RDF stores and datasets increases.

- **Single-version structured queries**: Structured queries performed on a specific version, not necessarily the newest one. In general, we may need to perform version materialization before processing this kind of query.
- **Cross-version structured queries**: Structured queries must be satisfied across different versions there by retrieving information common in many versions. Nonetheless, it adds a novel complexity and called also time-traversal queries.

2.3.2 Delta query

Delta query considers the changes occurring between two versions. We distinct *delta materialization*, *single-delta structured queries* and *cross-delta structured queries*.

- **Delta Materialization**: This query asks for the difference (delta) between two or more given versions. The deltas should be stored and computed. Nonetheless, the evaluation of this query would need the on-the-fly reconstruction of one or more of the non-materialized versions which engenders an overhead at query time .
- **Single-delta structured queries**: Structured queries are performed on a specific delta instance of the dataset. This information focuses particularly on the differences between two, not necessarily, consecutive versions.
- **Cross-delta structured queries**: Structured queries are evaluated on changes of several versions of the dataset.

2.4 Benchmarking RDF archiving systems

Towards evaluating the efficiency of RDF archives is essential to test the scalability of used storage techniques and the efficiency of the adopted querying strategies. Three benchmarks have been proposed in the literature for querying evolving

RDF data archives: BEAR (Fernández et al., 2015, 2016), EvoGen (Meimaris & Papastefanatos, 2016) and SPBv (Papakonstantinou et al., 2017).

2.4.1 BEAR RDF archiving benchmark

BEAR (Fernández et al., 2015, 2016; Fernández, Umbrich, Polleres, & Knuth, 2017) is the first benchmark proposed for querying evolving RDF data archives. BEAR is built in two systems using respectively Jena's TDB store [1] and HDT (Fernández, Martínez-Prieto, Gutiérrez, Polleres, & Arias, 2013). The authors propose a theoretical foundations for the design of benchmark queries and provides semantics for RDF archives. Four archiving strategies are implemented: independent copies, change based and time-based approaches in (Fernández et al., 2016) and recently an hybrid approach is introduced in the last version of the BEAR Benchmark. Five query operations are tested: version materialization, delta materialization, version queries, cross-version join and change materialization.
Despite the detailed theoretical formalization of RDF benchmarking problem is given by the BEAR system, a deep analysis must be done, in order to check out the challenges needed to improve the performance of an RDF archiving systems: the number of versions, the space needed to store a version, the type of query and the difficulties encountered when answered on top of more than one version. The aspect of optimization is not treated yet and a cost based model should be developed to decide if it will be more suitable to store a full version or a delta.
The BEAR system proposes a customizable generator for evolving synthetic RDF data making them scalable to any dataset size and number of versions. Nevertheless, more works need to be realized to provide queries corresponding to real user needs and to define more complex time-based queries.

2.4.2 EvoGen

Meimaris and al. have proposed a generator for evolving RDF data, named EvoGen Benchmark (Meimaris & Papastefanatos, 2016). EvoGen extends the LUBM ontology with 10 new classes and 19 new properties and implements a mechanism that produces logs of the changes between consecutive versions. EvoGen is configurable in terms of the archiving strategy (Independent copies, Deltas, etc.),

[1] https://jena.apache.org/dcoumentation/tdb/

the number of versions and the number of changes. An adaptive query workload is then implemented based on the evolution aspects of the data. The EvoGen ontolgy schema follows the DIACHRON model (Meimaris, Papastefanatos, Pateritsas, Galani, & Stavrakas, 2014), a graph based approach for annotating datasets with temporal information (time-based archiving approach). The system is equally configurable to other types of archiving strategies such as independent copies, delta-based and hybrid storage approaches.
The functionality of the system can be invoked through a Java API by importing EvoGen Libraries. A Version Management component is implemented to support different types of archives with a change materialization module responsible of creating a log of changes. The actual version of EvoGen provides adaptative query workload generator based on the LUBM queries extended with variables corresponding to a particular version or timeline.

2.4.3 SPBv

SPBv was developed in the context of the HOBBIT project[2] to test the ability of archiving system in order to manage efficiently evolving dataset and queries (Papakonstantinou et al., 2017). It is based on the Linked Data Benchmark Councils (LDBC)[3] Semantic Publishing Benchmark (SPB).
The SPBv is built on different components : a benchmark controller, a data generator, a task generator, an evaluation module and an evaluation storage. The benchmark controller is responsible of scheduling tasks to the data generator and the task generator.
The data generator uses seven core and three domains RDF ontologies for the data production. The task generator consists of several instances running in parallel, its main job to send the gold standard previously received from the data generator to the evaluation storage. Then the former component re sends the gold standard and the results reported by the benchmarke system to the Evaluation Module that is responsible for evaluating the performance of the system.

[2]https://project-hobbit.eu/
[3]ldbc.council.org

2.5 Discussion

Based on the state-of-the-art that we present in this chapter, we can conclude that the scalability problem is an important issue in the context of RDF archiving systems. In fact, based on a qualitative classification of the complexity (low, medium, high) required to satisfy each type of retrieval demand proposed in (Fernández, Polleres, & Umbrich, 2015), we find that the majority of query types present High or Medium level of complexity in IC and CB approaches (table 2.2). As the size of RDF data is continuously growing, managing RDF dataset archives may be seen as a Big Data problem. That is, it will be interesting to evaluate RDF archiving approaches using large scale frameworks namely Hadoop or SPARK. Based on the fact that the temporal aspect of the RDF archives makes them suitable for analysis and learning, we orient our choice to SPARK framework as it is more adapted in this context than Hadoop.

Policies			
RETRIEVAL NEED	Indep-Copies (IC)	Change-based (CB)	Time-stamp (TS)
Version Materialization	Low	Medium	Medium
Delta Materialization	Medium	Low	Low
Single-version structured queries	Medium	Medium	Medium
Cross-version structured queries	High	High	Medium
Single-delta structured queries	High	Medium	Medium
Cross-delta structured queries	High	High	Medium

Table 2.2: Processing of retrieval needs (level of complexity) (Fernández et al., 2015)

2.6 Conclusion

In this chapter, we first present the state-of-the-art of the approaches for managing evolving RDF dataset. We presented the basic strategies that archiving systems follow for storing multiple versions of a dataset, the different query types and we described the existing versioning Benchmarks along with their features and characteristics. In the next chapter, we present our approach which consists on using SPARK framework to evaluate RDF archiving strategies.

Part II
Contributions

Part II presents the contributions of this master thesis. Chapter 3 presents the theoretical aspects beyond the use of SPARK for managing and querying RDF dataset archives. Chapter 4 provides the experimental study realizing on the benchmark dataset.

Chapter 3

Proposed Approach: Evaluation of the RDF archiving strategies with SPARK

3.1 Introduction

The published RDF data in the Web is continuously evolving leading to an important number of RDF datasets in the Linked Open Data (LOD). There is thus an emergent need for efficient RDF data archiving systems. Obviously, the fast increasing size of RDF datasets raises the need to treat the problem of RDF archiving as a Big data problem. As we have seen in the last chapter, the proposed RDF archiving systems or benchmarks are built on top of existing RDF query processing engine. Nevertheless, no work has been realized for managing RDF archives on top of Big data processing frameworks. The problem is more challenging here as these frameworks are not designed for RDF processing nor for evolution management.
The objective of this chapter is to present the theoretical aspects beyond the use of SPARK for managing and querying RDF dataset archives: RDF datasets modeling and querying; how to use SPARK query engines (SPARK SQL or GraphX) for managing and querying RDF datasets versions; the rewriting of SPARQL into SPARK SQL/GraphX and finally the discussion of query optimization issues.
This chapter is structured as follows: in section 3.2, we propose a formal model

of RDF dataset archives and we propose a theoretical formalization of RDF versioning queries. In section 3.3, the principle aspects beyond the use of SPARK to manage and query RDF dataset archives are presented: from storing RDF datasets to querying and optimization issues.

3.2 A model for RDF dataset archives

In this section, we propose a formal representation of RDF dataset, RDF dataset archives and their relative versioning operations and we propose a theoretical formalization of RDF dataset versioning queries.

3.2.1 RDF Datasets

The Resource Description Framework (RDF) is a standard that supports the description of Web data resources. An RDF statement is a triple (`s`,`p`,`o`) where `s` is the subject, `p` the predicate and `o` the object of the statement used to describe resources and properties of the resources. More formally, given disjoint finite sets I, B and L (`IRIs`,`Blank nodes` and `Literals`), (`s`,`p`,`o`) $\in (I \cup B) \times I \times (I \cup B \cup L)$ is called an RDF triple.
An RDF dataset is a collection of RDF graphs. While RDF graph has a model-theoretic semantics which establishes when a model satisfies an RDF graph, no formal semantics exist for RDF dataset. An RDF dataset is a set of RDF resources URI associated with their description.

$$DS = \{(URI_1, (s_1, p_1, o_1)), ..., (URI_n, (s_n, p_n, o_n))\}$$

We can identify three possible changes between two versions of a Dataset:

- The creation of a new URI with an RDF description.
- The deletion of an URI-RDF couple.
- The update of the RDF description associated to a given URI.

As the update of an RDF description is considered as a deletion followed with the creation of a new URI-RDF couple, two possibles changes are considered in this paper: create and delete.

3.2.2 RDF Datasets versionning

An RDF triple in an RDF archive is associated to a tag which corresponds to the identification of its corresponding version. When we create an RDF triple we associate it to the number of the current version. An RDF dataset version is a set of RDF-URI couples associated with their version tags.

$$
\begin{aligned}
DS_{V_1} &= \{(URI_1, (s_1, p_1, o_1), V_1), ..., (URI_n, (s_n, p_n, o_n), V_1)\} \\
DS_{V_2} &= \{(URI_1, (s_1, p_1, o_1), V_2), ..., (URI_k, (s_k, p_k, o_k), V_2)\} \\
&... \\
DS_{V_m} &= \{(URI_2, (s_2, p_2, o_2), V_m), ..., (URI_k, (s_k, p_k, o_k), V_3)\}
\end{aligned}
$$

In version V_2, a new RDF URI, URI_k, is introduced. In version V_m, the URI_1 is deleted. Figure 3.1 shows an example of an evolving RDF dataset. In version V_2, the description of the URI_1 (e:toto,e:hasJob,e:analyst) is replaced with the description (e:toto e:hasJob e:dataSc). That is, the couple URI_1, RDF description of version V_1 is deleted and a new couple URI-RDF description is inserted. In version $V3$, the URI_2 is deleted and a new URI, URI_5 is inserted.

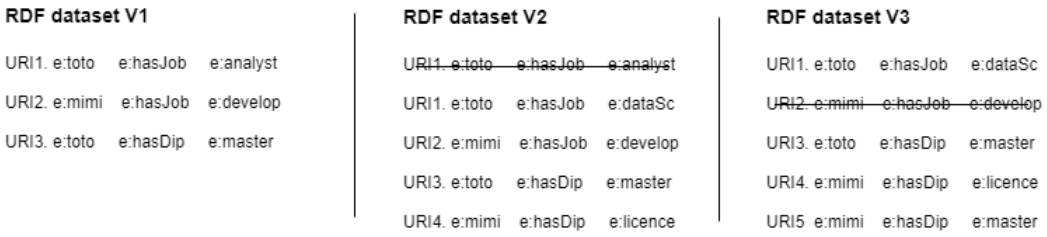

Figure 3.1: Example of RDF dataset versions.

The dataset versions of figure 3.1 are modeled as follows:

$$
\begin{aligned}
DS_{V_1} &= \{(URI_1, (e : toto, e : hasJob, e : analyst), V_1), (URI_2, (e : mimi, e : hasJob, e : develop), V_1), ...\} \\
DS_{V_2} &= \{(URI_1, (e : toto, e : hasJob, e : dataSc), V_2), (URI_2, (e : mimi, e : hasJob, e : develop), V_2), ...\} \\
DS_{V_3} &= \{(URI_1, (e : toto, e : hasJob, e : dataSc), V_3), (URI_3, (e : toto, e : hasDip, e : master), V_3), ...\}
\end{aligned}
$$

As we will see in the following section, the version tag is defined as an attribute in a SPARK table and we may query a version by indicating the number of its version in a SPARK SQL query.

3.2.3 RDF datasets archives querying

Querying evolving RDF datasets data represents the biggest challenge behind the use of archiving systems. We can classify six different types of retrieval needs regards the query type (materialization or structured query) and the main focus (version/delta) of the involved query and also we distinguish the time (single/cross-time queries) for the structured queries:

- **Version materialization:** A basic query in which a full version is retrieved. This kind of retrieval functionality presents a real challenge when the number of RDF triples increases. Depending on what we use as an archiving strategy, this query may either retrieve already fully materialized versions in case of Independent Copy based approach or reconstructs a specific version based on the associated changes or deltas for Change-based approach. Given a dataset archive $DA = \{DS_{V_1}, DS_{V_2}, ..., DS_{V_n}\}$, the version materialization of a dataset at a given version V_i is defined as follows:

$$Mat(V_i) = \{(URI, (s, p, o), V_i)\}$$

- **Delta materialization:** Theses queries are performed on two versions to detect the changes occurring at a given moment. Given a dataset archive DA, the delta materialization of the difference between two versions V_i and V_j of a dataset which corresponds to the RDF description $d = (URI, (s, p, o))$ that belong to V_i and do not belong to V_j and viceversa:

$$Delta(V_i, V_j) = \{(d, V_i) | (d, V_j) \notin V_j\} \cup \{(d, V_j) | (d, V_i) \notin V_i\}$$

- **Single-version query:** An RDF query performed on a specific version.

$$[[Q]]_{V_i} = \{(URI, (s, p, o), V_i) | \sigma_{s,p,o}(Q) \subseteq V_i\}$$

  where σ is a function which replaces eventual variables of query Q with values in (s, p, o).

- **Cross-version structured queries:** Also named time-traversal queries. These queries are evaluated across different versions.

$$Join(Q_1, V_i, Q_2, V_j) = [[Q1]]_{V_i} \bowtie [[Q2]]_{V_j}$$

3.3 RDF dataset archiving on Apache Spark

In this section, we present the main features of Apache SPARK cluster computing framework. We present the SPARK SQL query engine and we show how can we use it to formalize RDF versioning queries. Finally, we propose a formal presentation of the mapping of SPARQL to SPARK SQL.

3.3.1 RDF dataset storing Apache Spark

Apache Spark (Zaharia et al., 2016b) is a main-memory extension of the MapReduce model for parallel computing that brings improvements through the data-sharing abstraction called Resilient Distributed Dataset (RDD) (Zaharia et al., 2012) and Data frames offering a subset of relational operators (*project*, *join* and *filter*) not supported in Hadoop. Spark also offers two higher-level data accessing models, an API for graphs and graph-parallel computation called GraphX (Graube et al., 2014) and Spark SQL, a Spark module for processing semi-structured data. As the RDF data model is interpreted as a graph and processing SPARQL can be seen as a subgraph query pattern matching, GraphX seems to offer a natural way for querying RDF data (Schätzle, Przyjaciel-Zablocki, Berberich, & Lausen, 2015). Nevertheless, GraphX is optimized to distribute the workload of highly-parallel graph algorithms, such as PageRank, that are performed on the whole graph. However, this process is not adapted for querying RDF datasets where queries define a small subgraph pattern leading to highly unbalanced workloads (Schätzle et al., 2015; Naacke et al., 2016).
SPARK SQL (Armbrust et al., 2015) is a SPARK module that performs relational operations via a DataFrame API offering users the advantage of relational processing, namely declarative queries and optimized storage. SPARK SQL supports relational processing both on native RDDs or on external data sources using any of the programming language supported by SPARK, e.g, Java, Scala or Python (Armbrust et al., 2015). SPARK SQL can automatically infer their schema and data types from the language type system. Equally, SPARK SQL allows for querying semi-structured and supports query federation allowing relational operations to be performed on disparate sources. All these features are built on the Catalyst framework which is claimed to be a highly extensible optimizer for SPARK SQL query processing (Armbrust et al., 2015).

3.3.2 RDF dataset querying with Spark SQL

SPARK SQL offers the users the possibility to extract data from heterogeneous data sources and can automatically infer their schema and data types from the language type system (e.g Scala, Java or python). In our approach, we use SPARK SQL for querying and managing the evolution of Big RDF dataset. An RDF dataset stored in HDFS is mapped into a SPARK Dataframes (equivalent to tables in a relational database) with columns corresponding respectively to the subject, property, object and eventually a tag of the corresponding version.
Figure 3.2 shows a view of a part of a dataframe containing two versions of the RDF dataset defined on the example used in the previous section.

Subject	Predicate	Object	Version
e:toto	e:hasJob	e:analyst	V1
e:mimi	e:hasJob	e:develop	V1
e:toto	e:hasJob	e:dataSc	V2
e:mimi	e:hasJob	e:develop	V2

Figure 3.2: Example of a Dataframe with RDF dataset attributes

In order to obtain a view of a dataframe named “table”, for example, we execute the following SPARK SQL query:

SELECT * FROM table

When we want to materialize a given version, V_1 for example, the following SPARK SQL query is used:

SELECT Subject,Object,Predicate FROM table WHERE Version ='V_1'

Another advantage beyond the use of SPARK SQL for the RDF dataset archiving is the scalability of RDF change detection issue. Many approaches implement change detection algorithms on the MapReduce framework (Ahn, Im, Eom, Zong, & Kim, 2014). Using SQL SPARK, we can easily detect the change between two different versions by executing a simple SQL SPARK query:

SELECT Subject,Predicate,Object FROM table WHERE Version='V_i'
MINUS
SELECT Subject,Predicate,Object FROM table WHERE Version='V_j'

Using SPARK SQL, we can define RDF dataset archiving queries as follows:

- **Version materialization**: $Mat(V_i)$.

  SELECT Subject,Object,Predicate FROM table WHERE Version ='V_i'

- **Delta materialization**: $Delta(V_i, V_j)$.

  SELECT Subject,Predicate,Object FROM table WHERE Version='V_i'
  MINUS
  SELECT Subject,Predicate,Object FROM table WHERE Version='V_j'
  UNION
  SELECT Subject,Predicate,Object FROM table WHERE Version='V_j'
  MINUS
  SELECT Subject,Predicate,Object FROM table WHERE Version='V_i'

- **Single-version query**: $[[Q]]_{V_i}$. We suppose here a simple query Q which asks for all the subject in the RDF dataset (mapping a SPARQL query into a SPARK SQL query is an important issue and will be treated separately in the next section).

  SELECT Subject FROM table WHERE Version='V_i'

- **Cross-version structured query**: $Join(Q_1, V_i, Q_2, V_j)$. We suppose that the two queries concerns as a part the subject column. What we need here is a join between the two query results. We define two dataframe table$_i$ and table$_j$ containing respectively the version V_i and V_j. The cross-version query is defined as follows:

  SELECT * FROM table$_i$
  INNER JOIN table$_j$
  ON table$_i$.Subject = table$_j$.Subject

We note that, for more clarity, we have supposed in this section the use of simple single-version and cross-version queries. Nevertheless, in real world applications, complex SPARQL query are used and a mapping from SPARQL query to SPARK SQL is needed.

3.3.3 From SPARQL to SPARK SQL

SPARQL is a subgraph matching query language. A *Select* SPARQL query is of the form (W, P), where W is a finite set of variables and P is a *graph pattern*. The evaluation of a query pattern P on an RDF graph G consists on subgraph matching of the graph pattern P against the graph G based on a set of mappings σ from W to the terms of G. This is can be formally defined as follows (Arenas, Gutierrez, & Pérez, 2009):

$$[[P]]_G = \{\ \sigma : W \longrightarrow T \mid \text{domain}(\sigma) = \text{var}(P) \cap W \text{ and } \sigma(P) \subseteq G\ \}$$

A SPARQL graph pattern is defined recursively as follows:

- A triple pattern t is a graph pattern.
- If P_1 and P_2 are graph patterns then (P_1 AND P_2) and (P_1 UNION P_2) are graph patterns.
- If P is a graph pattern and R a value constraint then (P FILTER R) is a graph pattern.

SPARK SQL is used in (Naacke et al., 2016; Schätzle et al., 2016) for querying RDF big data where a query compiler from SPARQL to SPARK SQL is provided. That is, a FILTER expression can be mapped into a condition in Spark SQL while UNION, OFFSET, LIMIT, ORDER BY and DISTINCT are mapped into their equivalent clauses in the SPARK SQL syntax.
SPARQL graph pattern can have different shapes which can influence query performance. Depending on the position of variables in the triple patterns, SPARQL graph pattern may be classified into three shapes (figure 3.3):

1. Star pattern: this query pattern is commonly used in SPARQL. A star pattern has diameter (longest path in a pattern) one and is characterized by a subject-subject joins between triple patterns.
2. Chain pattern: this query pattern is characterized by object-subject (or subject-object) joins. The diameter of this query corresponds to the number of triple patterns.
3. Snowflake pattern: this query pattern results from the combination of many star patterns connected by short paths.

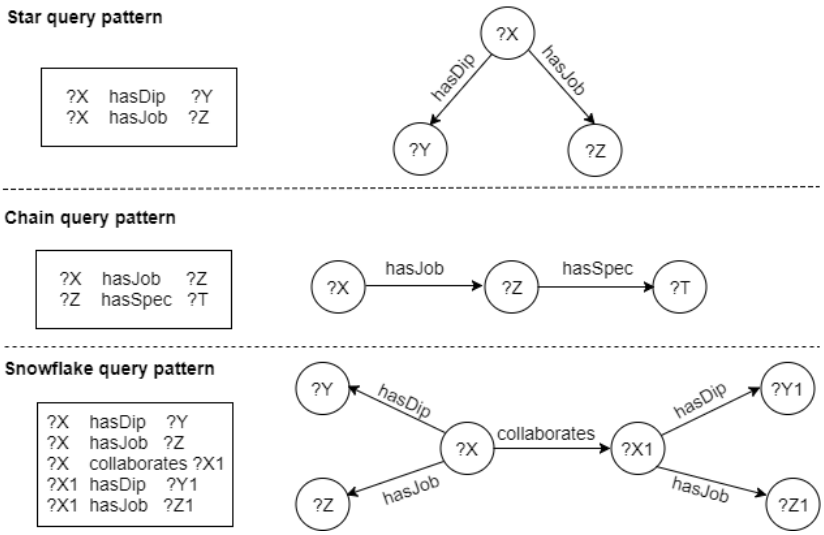

Figure 3.3: SPARQL graph pattern shapes.

When we query RDF dataset archives, theses query shapes concerns single version ($[[Q]]_{V_i}$) and cross-version queries ($Join(Q_1, V_i, Q_2, V_j)$) where Q may have one of the defined shapes. We show in the following how we proceed for different SPARQL query patterns:

- Star pattern: given a SPARQL query pattern (?X, hasDip ?Y, ?X hasJob ?Z), we need to create two dataframes df_1 and df_2 as follows:

  df_1= "SELECT Subject, Object FROM table
  WHERE Predicate = 'hasDip'"
  df_2= "SELECT Subject, Object FROM table
  WHERE Predicate = 'hasJob'"

  The query result is obtained as a join between dataframes df_1 and df_2:

  SELECT * FROM df_1
  INNER JOIN
  df_2 ON df_1.Subject = df_2.Subject

- Chain pattern: given a SPARQL query chain with two triples (?X, hasJob ?Z, ?Z hasSpec ?Z), we need to create a dataframe for each triple:

  df_1= "SELECT Subject, Object FROM table
  WHERE Predicate = 'hasJob'"
  df_2= "SELECT Subject, Object FROM table
  WHERE Predicate = 'hasSpec'"

The query result is obtained as a join between dataframes df_1 and df_2:

SELECT * FROM df_1
INNER JOIN
df_2 ON df_1.Object = df_2.Subject

We note that here the number of join operations depends on the number of query triples. A chain query with n triples t_i needs n-1 joins:

$$join(join(join(join(t_1, t_2), t_3), t_4), ..., t_n)$$

- Snowflake pattern: the rewritten of snowflake queries follows the same principle and may need more join operations depending equally on the number of triples used in the query.

For single version query $[[Q]]_{V_i}$, we need to add a condition on the version for which we want to execute the query Q. Nevertheless, the problem becomes more complex for cross-version join query $Join(Q_1, V_i, Q_2, V_j)$ as other join operations are needed between different versions of the dataset. Two cases may occur:

1. Cross-version query $type_1$: this type of cross-version queries concerns the case where we have one query Q on two or more different versions. For example, to follow the evolution of a given person career, we need to execute (?X hasJob ?Z) on different versions. Given a query Q and n versions, we denote T_1,...,T_n the different results obtained by executing Q on versions V_1,...,V_n. The final result is obtained by realizing the union of the different T_i. What we can conclude here is that the number of versions does not increase the number of joins which depends only on the shape of the query.

2. Cross-version query $type_2$: the second case occurs when we have two or more different queries $Q1$, Q_2,...,Q_m on many different versions. For example, we may need to know if the diploma of a person ?X has any equivalence in RDF dataset archive:

   Q_1 :?X hasDip ?Y on version V_1
   Q_2 :?Y hasEqui ?Z on versions V_2, ..., V_n

   Given T_1,...,T_n the different results obtained by executing $Q1$, Q_2,...,Q_n, respectively, on versions V_1,...,V_n, the final result is obtained with a combination of join and/or union operations between the T_i. In the worst case we may need to compute n-1 joins:

   $$join(join(join(join(T_1, T_2), T_3), T_4), ..., T_n)$$

That is, for cross version query $type_2$, the number of joins depends on the shape of the queries as well as the number of versions.

3.4 Data partitioning and query optimization

In this section, we define data partitioning and we discuss its use in the context of RDF datasets. We present then the principle that we adopt to partition RDF datasets for efficiently executing: version materialization, delta materialization, single version and cross versions queries. Then, we present Spark Optimizer Catalyst and we discuss its use in our context.

3.4.1 Data partitioning

Ihe principle that we adopt for the partition of the data in case of single version and cross-version queries (figure 3.4) is presented as follows:

- First of all, we load two RDF files (version V_1 and version V_2) from HDFS as input. The input RDF files are assumed to be in a N-triple format that represents one triple in a single line.
- Then, a mapping steps from RDF files into dataframes with corresponding columns Subject, Object, Predicate and a tag of the version.
- We adopt a partitioning by RDF subject for each version.
- The SPARK SQL engine processes the partitioning parts stored in relational database and the query result is returned.

Concerning Version and delta materialization queries, we adopt the same process without partitioning the data (figure 3.5). In fact, as all the data (version or delta) will be loaded, no partition is needed.

3.4.2 Spark Optimizer

SPARK optimizer Catalyst generates multiple physical plans and compares them based on cost. In the actual version of SPARK SQL (Armbrust et al., 2015), Select

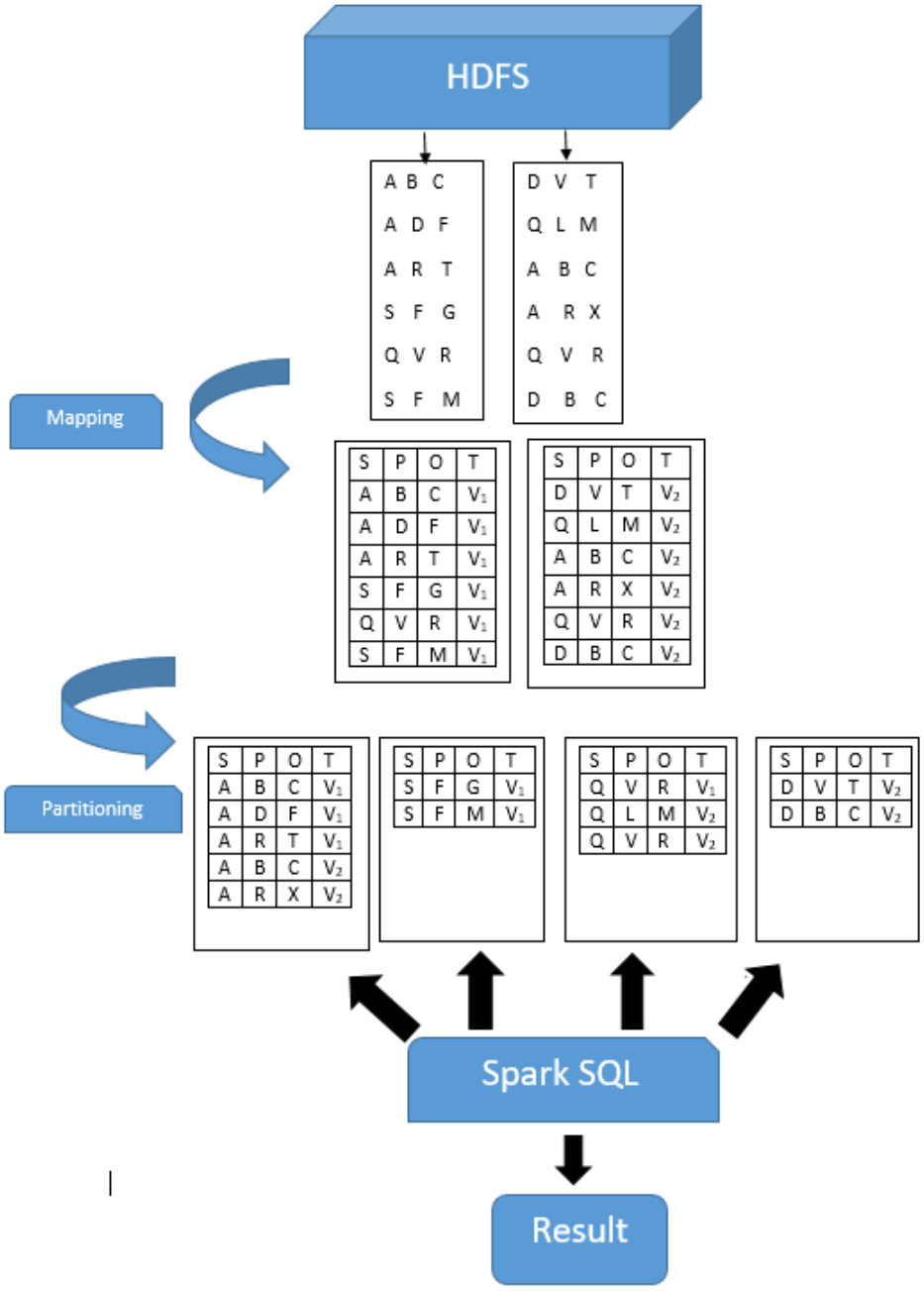

Figure 3.4: Query execution with data partition of single version and cross-version queries.

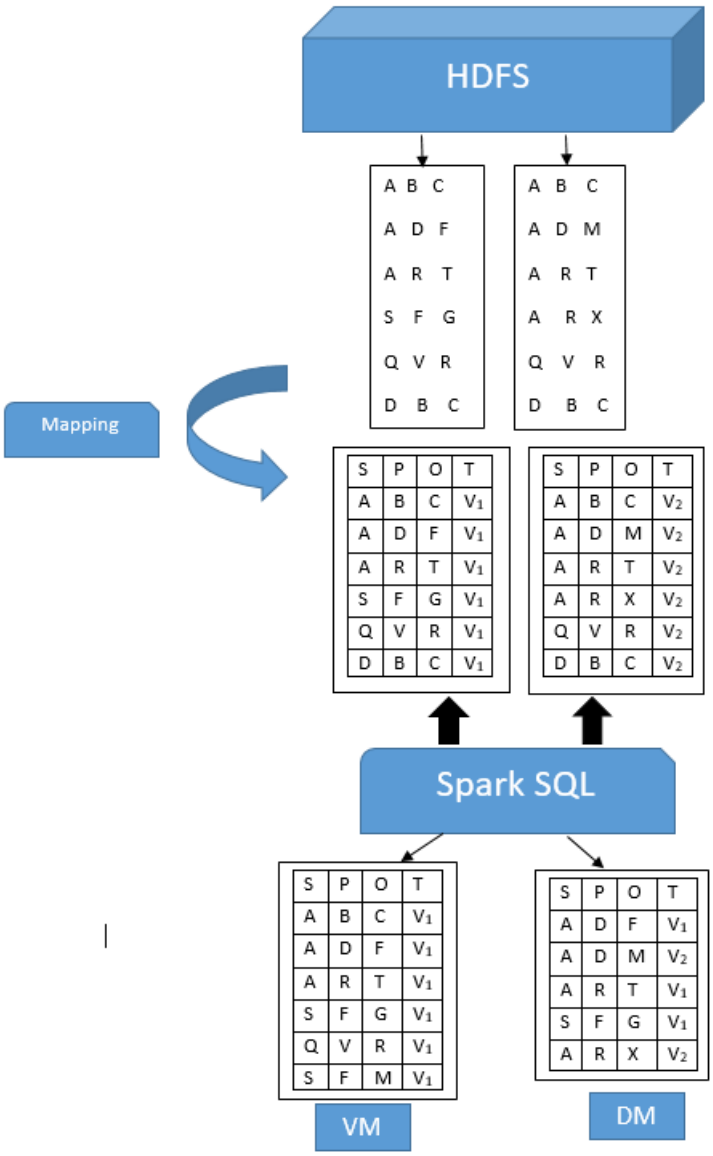

Figure 3.5: Query execution without data partition of Version and Delta materialization queries.

join queries benefit from cost-based optimization and broad cast join are used for small relations. That is, given a chain pattern SPARQL query (t_1,t_2,t_3), Catalyst generates a physical plan by computing a cross product between t_1 and t_3 before

joining with t_2.
Catalyst optimizer may be extended with user defined rules in order to fulfill more efficient query execution plans (Armbrust et al., 2015). We believe that many works can be done on this context if we want to adapt the execution plan by taking into consideration, the size of a version, the number of versions and the shape of used queries. In the actual status of our work, we use Catalyst without any extension.

3.5 Conclusion

In this chapter, we present the main contribution of our master thesis which consists on the use of SPARK to manage and query RDF dataset archives. After giving a formal modeling of RDF dataset archives and change operations, we propose a theoretical formalization of RDF versioning queries and we propose a formal rewritten of theses queries with SPARK SQL. What we can conclude is that many criteria have to be considered when we realize experimentation: the versioning strategy, the types of versioning queries, the shape of SPARQL queries, the number of versions and eventually, the partitioning strategy. The evaluation and the interpretation of the experimental results given in chapter 4 provides an effective idea about the works that we may need to do for query optimization as we can see the real impact of Catalyst execution plan on the query performance.

Chapter 4

Experimental study

4.1 Introduction

In order to evaluate the RDF archiving systems, we performed different types of queries. We present in this chapter an experimental study realized on benchmark datasets. The effectiveness of the proposed approach is evaluated in terms of running query time .
This chapter is organized as follows: Section 4.2 details experimental environment. Section 4.3 describes the evaluation criteria to test the performance of our proposed approach and finally, section 4.4 presents experimental results obtained on the different kind of queries.

4.2 Environment

For fairness, the experiments are performed on single machine in local mode where the node has 4 cores (Intel Corei7-7500U Proceesor up to 3.5ghz), 12GB of RAM and 1T of disk. Experiments were realized using Apache Spark version 2.1.1, Apache Hadoop version 2.7.3 and Ubuntu 16.04 and all experiments were implemented in scala version 2.1.1. We use the synthetic data (N-triplet) of the Benchmark BEAR[1].

[1]https://github.com/webdata/BEAR

Apache Spark propose four programming APIs: Scala, Java, Python and recently R. The reason why we choose Scala is because Scala is a powerful language adopted by several developers and researchers especially when we are talking about Big data tools (Odersky et al., 2004). Equally, Spark is written in Scala so knowing Scala coding make it easy for us to understand Spark functionalities and to modify, if needed, some of them.

4.3 Dataset Description

As mentioned in the previous section, we are using the BEAR dataset. RDF archive data monitor more than 650 different domains across time and composed of 58 snapshots, i.e. 58 versions. Each snapshot consists of triples annotated with their RDF document provenance in N-triple format. Figure 4.1 describes the number of statements of each version in different policies: Independent Copies and Change based. Each snapshot consists of roughly 500m triples, the adds and deletes statements are also depicted in the same figure.

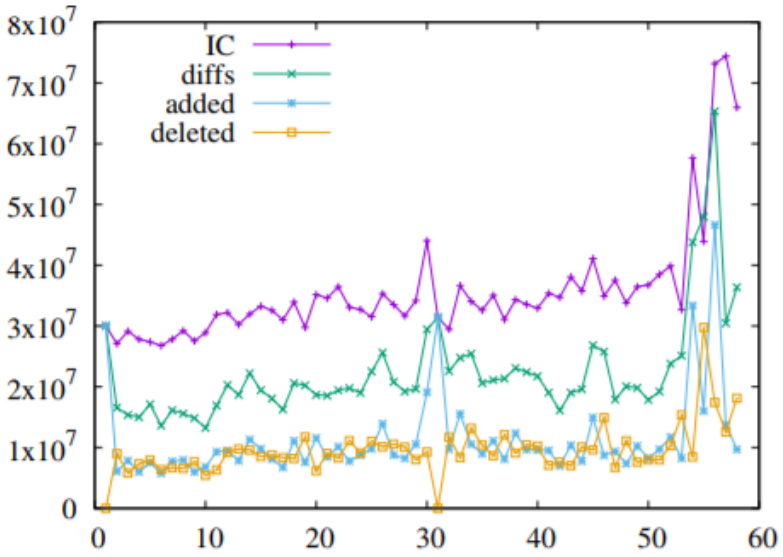

Figure 4.1: Number of statements (Fernández et al., 2017)

4.4 Evaluation criteria

The evaluation criteria used for experimentation are defined as follows:

- Versioning strategies: change based or independent copy.
- Types of versionning queries: version materialization, delta materialization, single version and cross-version queries.
- The shape of SPARQL queries: Star or Chain.
- Number of versions: from 1 to 58 versions.
- Partitioning: the data was hash partitioned by RDF subject across multiple nodes.

4.5 Query evaluation

In the following we present the evaluation of versionning queries on top of SPARK framework. The evaluation concerns four query types: Version materialization, delta materialization, single version and cross version queries respectively.

4.5.1 Version Materialization

The content of the entire version is materialized. For each version, the average execution time of the queries was computed. Plots presented in figure 4.2 show the execution times of version materialization queries with Independent Copy (IC) and Change-based (CB) archiving strategies. The obtained results show that the execution times obtained with IC strategy are approximately constant with a small variation caused by the changes occurring on the size of the used versions. Equally, the execution times of IC approach show better result then the ones obtained with CB approach. In fact, versions in CB approach are not already stored and need to be computed each time we want to query a given version.

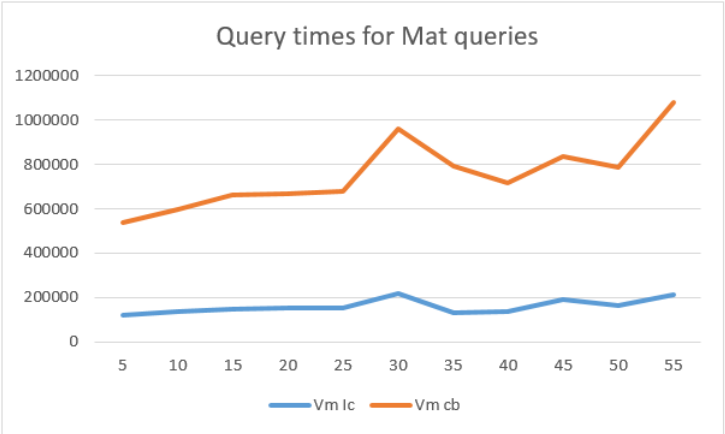

Figure 4.2: Version Materialization queries

4.5.2 Delta Materialization

In order to observe real differences between versions, we use an increasing interval of 5 versions between the ones for which we want to compute the deltas. That, is we compute $Delta(V_0, V_i)$ where $i \in \{5, 10, 15, , 50, 55.\}$. Based on the plots shown in figure 4.3 , we may observe that the execution times obtained with IC strategy are approximately constant and show better results compared to the ones obtained with CB approach. In fact, given a query $Delta(V_0, V_i)$ executed with CB approach, the difference between the two versions are not necessarily stored and need to be computed on the fly. As the version V_i is not stored we need to be compute it based on stored deltas between V_0 and V_i.

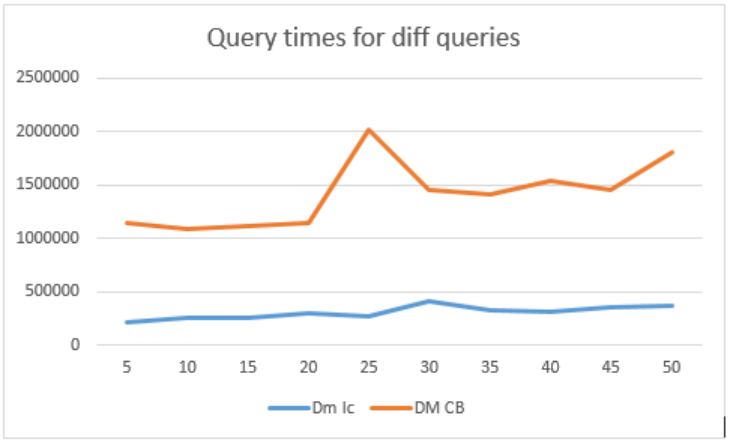

Figure 4.3: Delta Materialization queries

4.5.3 Single-version queries

We use only query with one triple pattern (pattern shapes are evaluated with cross-version queries) and we make different variations with subject, predicate and object based queries. Before giving the query execution times, figure 4.4 presents the carnality's of single version query results in order to track the evolution of the used dataset.

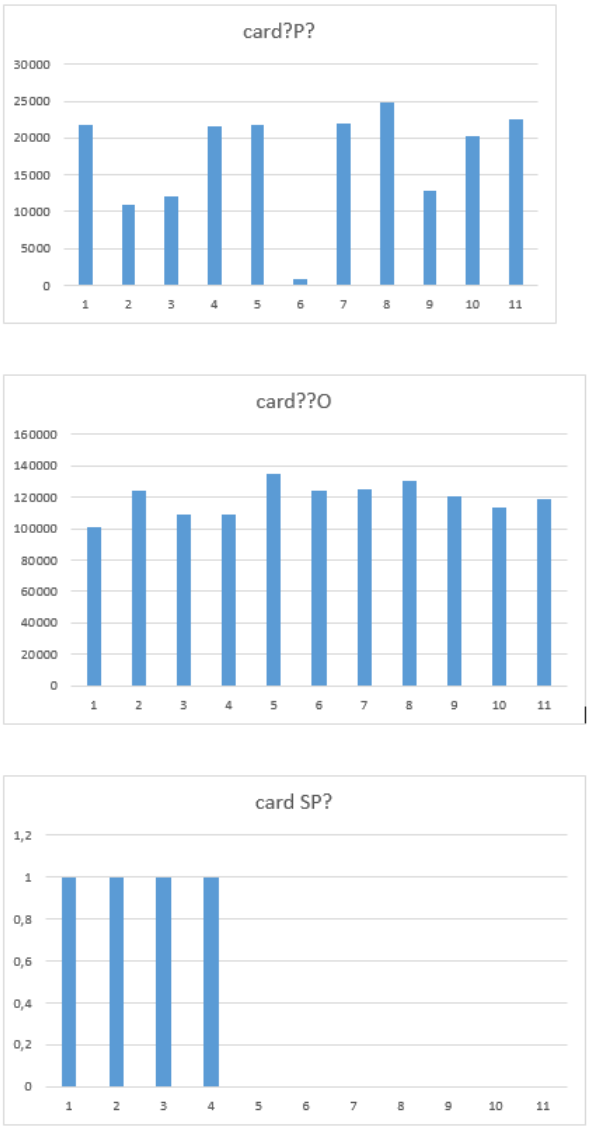

Figure 4.4: An example of some queries results: Card

Concerning execution times, The obtained results are shown in figure 4.5 and 4.6 respectively. Figure 4.5 concerns single version queries where the subject is given in the query whereas the object and/or the predicate correspond to what we ask for.

Figure 4.5: Single version queries (Subject)

In the other hand, figure 4.6 concerns single version queries where the object and/or the predicate are given in the query whereas the subject is what we ask for. The queries are executed on versions with different sizes. The analysis of the obtained plots shows that the use of partitioning ameliorates considerably the query execution times. Equally, as no join operations are needed for this kind of queries, the use of partition does not variate with the size of the used versions and is approximately constant. As we have seen in Chapter 3, SPARQL queries may have different shapes (Star, Chain and snowflake) inducing a certain number of join operations. In the following section, we will observe how the execution of theses queries will be affected with the size/number of versions. Equally, we will observe to what extend the use of partition allow as to overcome this issue.

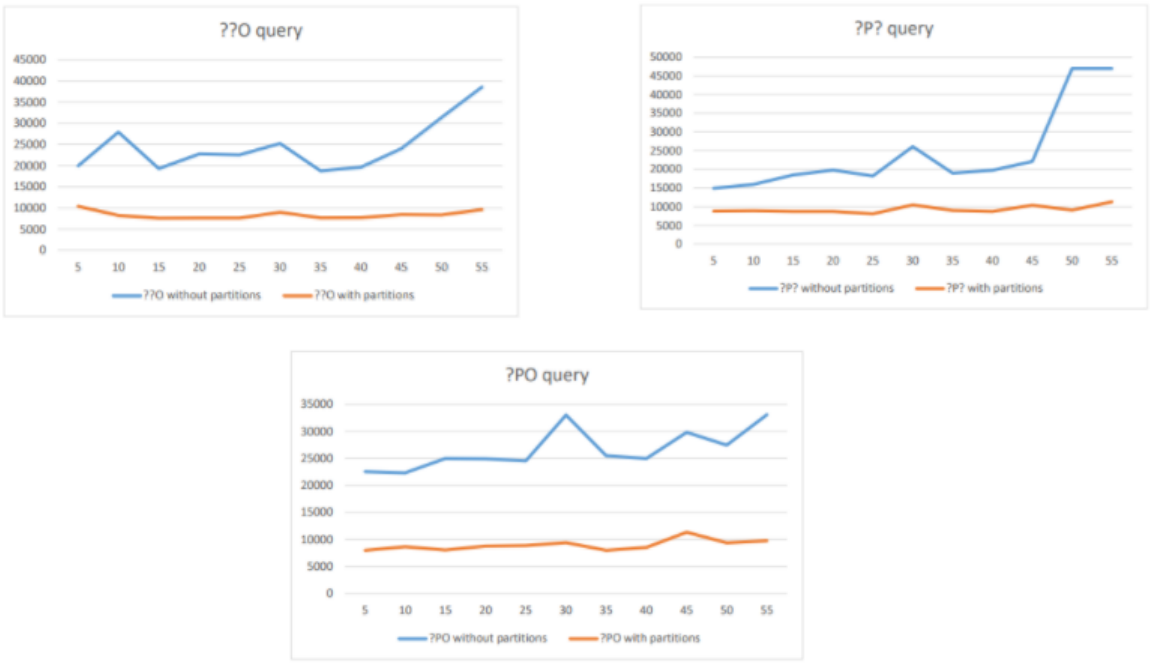

Figure 4.6: Single version queries (Object and predicate)

4.5.4 Cross-version queries

In this section, we focus on Cross-version queries and we present a series of tests realized on Star and Chain query shapes. Figure 4.7 concerns Star query shape and 4.8 show the obtained execution times for Chain queries. As we can observe, the use of partitioning ameliorates execution times for Star and Chain queries.

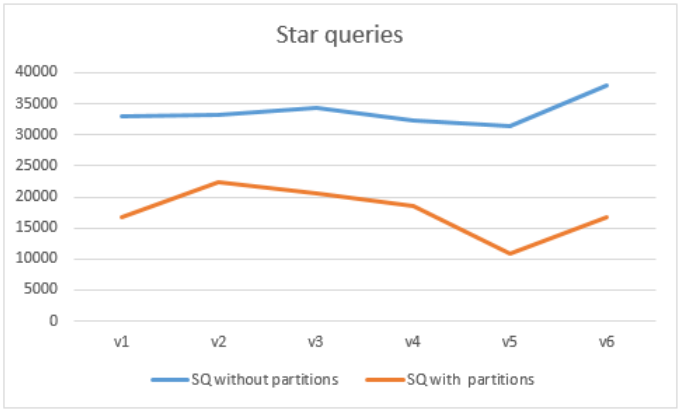

Figure 4.7: Cross-version queries: Star query shape

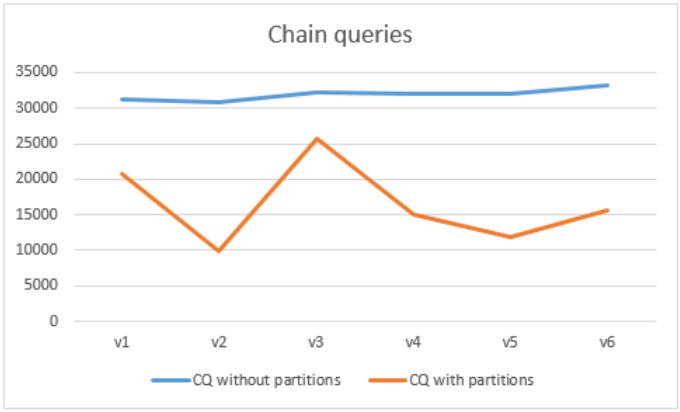

Figure 4.8: Cross-version queries: Chain query shape

We give in table 4.1 and table 4.2 an overview of the execution times obtained with Star and Chain queries respectively. As we can see, the obtained execution times is deeply ameliorated, for the two query shapes, when we use partition. Nevertheless, eventhough we use partitions, the execution times obtained with Star/Chain queries drop or fall depending on the size of the version. The variation of execution times is not the same for the two query shapes.

Versions	Triples	SQ without partitions (ms)	SQ with partitions (ms)
Version 1	30,035,245	23644.358	7399.109
Version 2	27,100,720	26232.014	7806.819
version 3	29,152,464	25864.583	7194.007
version 4	27,827,475	23441.38	6949.075
version 5	27,377,065	28657.241	7297.64
version 6	26,765,320	23267.585	7851.598

Table 4.1: Query evaluation performance for Star query (SQ)

As we can see in table 4.1, the execution times obtained for Star queries do not vary considerably with the size of the versions (figure 4.6). This is due to the fact that for Star queries, the number of joins operations do not increase with the size of the version.

Concerning Chain queries, we can see that the execution times deeply increase when we rise up the size of the version (table 4.2). In fact, as we have seen in Chapter 3, the execution of Chain queries needs more join operations then the execution of Star shape ones.

Versions	Triples	CQ without partitions (ms)	CQ with partitions (ms)
Version 1	30,035,245	31168.411	20743.833
Version 2	27,100,720	30762.349	9979.434
version 3	29,152,464	32297.495	25697.606
version 4	27,827,475	31969.248	15023.631
version 5	27,377,065	32080.286	11793.13
version 6	26,765,320	33147.345	15540.856

Table 4.2: Query evaluation performance for chain query (CQ)

4.6 Conclusion

In this chapter, we described the experimental environment then we enumerated the different evaluation criteria and we performed several experiments on different kind of queries. We conclude that the efficiency of our proposed approach is satisfied through the use of partitioning by RDF subject in case of single-version and cross-version query. Nevertheless, when we use complex queries (e.g Chain query shape), the execution times obtained with partitions are affected with the size of the used versions. This issue may be resolved by using other partition strategies by taking into consideration the query pattern and/or by extending the execution plan of Spark Catalyst optimizer.

Conclusion

The primary focus of this master thesis was studying the preliminaries and the state-of-art approaches for managing and benchmarking evolving RDF data. We presented the basic strategies that archiving tools follow for storing multiple versions of a dataset and we described the existing archiving benchmarks along with their features and characteristics. Therefore, the number and the size of RDF dataset keep growing at a fast pace and confronts the problem of RDF archiving as a Big data problem.

In this context, we proposed an evaluation of RDF dataset archiving strategies with SPARK which is designed for managing and querying RDF data archives using both Independent Copies (IC) and Change based (CB) approaches. Besides, we defined a theoretical formalization of RDF versioning queries using SPARK SQL syntax. We implemented the state of art approaches policies with spark scala. Several criteria has been considered to evaluate the performance of RDF archiving systems basically the versioning approaches (IC or CB), the types of RDF archives queries (version materialization, delta materialization, single version and cross version), the number of versions, the shape of SPARQL queries and the data partitioning strategy.

Results clearly confirm that the use of partitioning in SPARK shows an amelioration of the execution times compared with the results obtained without partitioning. Nevertheless, when we use complex queries (e.g Chain query shape), the execution times obtained with partitions are affected with the size of the used versions.

As a future work, we consider that cross-version queries are an important query type that need more investigation. Queries corresponding to real users needs have to be defined. In fact, analyzing the evolution of data across time is an impor-

tant issue and does not need to be proved. That is, if we want to analyze the evolution of data or to compare different data evolution shapes or behaviour over time, we need to look forward the amelioration of the execution time. Different issues need to be considered. Among other, the partitioning strategy (Schätzle et al., 2016), the execution plan of join operations (Naacke et al., 2016) and a cost-based model will be useful to re-partition the data for more complex queries (chain or snowflake) (Naacke et al., 2016).

Appendix

In this appendix, we will present the different RDF serialization formats.

Appendix A

RDF serialization formats

In this Appendix, we introduce a family of alternative text-based RDF serializations whose members have the same origin, but balance differently between readability for humans and machines.

N-triples

The N-Triple notation is very straightforward, each line of output in N-Triple format represents a single statement containing a subject, predicate and object. It is written as separate line where the URI is written between angle brackets ⟨*and*⟩ and terminated by a period (.) The files with N-Triples have the .nt extension.The figure A.1 represent the N-Triple example format.

01. <http://www.disco.unimib.it/go/45827> <http://dbpedia.org/ontology/birthDate> "1986-10-09"^^ <http://www.w3.org/2001/XMLSchema#date>
02. <http://www.disco.unimib.it/go/45827> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person>
03. <http://www.disco.unimib.it/go/45827> <http://dbpedia.org/ontology/affiliation> <http://www.disco.unimib.it/>
04. <http://www.disco.unimib.it/go/45827> <http://xmlns.com/foaf/0.1/name> "Blerina Spahiu"@en
05. <http://www.disco.unimib.it/go/45827> <http://dbpedia.org/ontology/author> <http://ceur-ws.org/Vol-1605/paper3.pdf>

Figure A.1: RDF/N-triples example format (Spahiu, 2017)

RDF/XML

The RDF/XML is one of the format for representing RDF triples .It defines a normative syntax for serializing RDF graphs as XML documents.It is built up from a series of smaller descriptions, each of which traces a path through an RDF graph. These paths are described in terms of the nodes (subjects) and the links (predicates) that connect them to other nodes (objects).The figure A.2 represent the RDF/XML example format .

```
01. <?xml version="1.0" encoding="UTF-8"?>
02.
03. <rdf:RDF
04.       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
05.       xmlns:dc="http://purl.org/dc/elements/1.1/"
06.
07.       <rdf:Description rdf:about="https://en.wikipedia.org/wiki/Milan">
08.             <dc:title>Milan</dc:title>
09.             <dc:publisher>Wikipedia</dc:publisher>
10.             <region:population>10000000</region:population>
11.             <region:principaltown rdf:resource="https://it.wikipedia.org/wiki/Lombardia"/>
12.       </rdf:Description>
13.
14. </rdf:RDF>
```

Figure A.2: RDF/XML example format (Spahiu, 2017)

N3

N3 is a language for expressing data and rules. It extends RDF with features such as variables, universal and existential quantification. The RDF/N3 example format is shown in figure A.3.

Turtle

An Rdf sysntax is a text-based serialization format. It is an extension of N-Triples.

```
01. @prefix dbo: <http://dbpedia.org/resource/> .
02. @prefix foaf: <http://xmlns.com/foaf/0.1/> .
03. @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
04. @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#
05.
06. dbo:George_Clooney rdf:type foaf:Person;
07. foaf:age ”55"^^xsd:int;
08. foaf:birthday "1961-06-05"^^xsd:date;
09. foaf:name "George Clooney"@en .
```

Figure A.3: RDF/N3 example format (Spahiu, 2017)

A.1 Conclusion

In this Appendix, we have presented the different serialization format.In our work,we are using the N-triple format for managing and querying RDF archive dataset.

References

Ahn, J., Im, D., Eom, J., Zong, N., & Kim, H. (2014). G-diff: A grouping algorithm for RDF change detection on mapreduce. In *Semantic technology - 4th joint international conference, JIST 2014, chiang mai, thailand, november 9-11, 2014. revised selected papers* (pp. 230–235).

Anuradha, J., et al. (2015). A brief introduction on big data 5vs characteristics and hadoop technology. *Procedia Computer Science*, *48*, 319–324.

Arenas, M., Gutierrez, C., & Pérez, J. (2009). On the semantics of SPARQL. In *Semantic web information management - A model-based perspective* (pp. 281–307).

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., ... Zaharia, M. (2015). Spark SQL: relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data, melbourne, victoria, australia, may 31 - june 4, 2015* (pp. 1383–1394).

Bizer, C., Heath, T., & Berners-Lee, T. (2009). Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, *5*(3), 1–22.

Broekstra, J., Kampman, A., & van Harmelen, F. (2002). Sesame: A generic architecture for storing and querying RDF and RDF schema. In *The semantic web - ISWC 2002, first international semantic web conference, sardinia, italy, june 9-12, 2002, proceedings* (pp. 54–68).

Cassidy, S., & Ballantine, J. (2007). Version control for RDF triple stores. In *ICSOFT 2007, proceedings of the second international conference on software and data technologies, volume isdm/ehst/dc, barcelona, spain, july 22-25, 2007* (pp. 5–12).

Chen, C. L. P., & Zhang, C. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Inf. Sci.*, *275*, 314–347.

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

de Sompel, H. V., Sanderson, R., Nelson, M. L., Balakireva, L., Shankar, H., & Ainsworth, S. (2010). An http-based versioning mechanism for linked data. *CoRR*.

Fernández, J. D., Martínez-Prieto, M. A., Gutiérrez, C., Polleres, A., & Arias, M. (2013). Binary RDF representation for publication and exchange (HDT). *J. Web Sem.*, *19*, 22–41.

Fernández, J. D., Polleres, A., & Umbrich, J. (2015). Towards efficient archiv-

ing of dynamic linked open data. In *Proceedings of the first DIACHRON workshop on managing the evolution and preservation of the data web co-located with 12th european semantic web conference (ESWC 2015), portorož, slovenia, may 31, 2015.* (pp. 34–49).

Fernández, J. D., Umbrich, J., & Polleres, A. (2015). *Bear: Benchmarking the efficiency of rdf archiving* (Tech. Rep.). Department fr Informationsverarbeitung und Prozessmanagement, WU Vienna University of Economics and Business.

Fernández, J. D., Umbrich, J., Polleres, A., & Knuth, M. (2016). Evaluating query and storage strategies for rdf archives. In *Proceedings of the 12th international conference on semantic systems* (pp. 41–48). New York, NY, USA: ACM.

Fernández, J. D., Umbrich, J., Polleres, A., & Knuth, M. (2017). Evaluating query and storage strategies for rdf archives. *Semantic web journal IOS Press*.

Graube, M., Hensel, S., & Urbas, L. (2014). R43ples: Revisions for triples - an approach for version control in the semantic web. In *Proceedings of the 1st workshop on linked data quality co-located with 10th international conference on semantic systems, ldq@semantics 2014, leipzig, germany, september 2nd, 2014.*

Gutierrez, C., Hurtado, C. A., & Vaisman, A. A. (2007). Introducing time into RDF. *IEEE Trans. Knowl. Data Eng.*, *19*(2), 207–218.

Huang, J., Abadi, D. J., & Ren, K. (2011). Scalable SPARQL querying of large RDF graphs. *PVLDB*, *4*(11), 1123–1134.

IM, D.-H., LEE, S.-W., & KIM, H.-J. (2012). A version management framework for rdf triple stores. *International Journal of Software Engineering and Knowledge Engineering*, *22*(01), 85-106.

Meimaris, M., & Papastefanatos, G. (2016). The evogen benchmark suite for evolving rdf data. In *Mepdaw/ldq@eswc.*

Meimaris, M., Papastefanatos, G., Pateritsas, C., Galani, T., & Stavrakas, Y. (2014). Towards a framework for managing evolving information resources on the data web. In *Proceedings of the 1st international workshop on dataset profiling & federated search for linked data co-located with the 11th extended semantic web conference, profiles@eswc 2014, anissaras, crete, greece, may 26, 2014.*

Memishi, B., Ibrahim, S., Pérez, M. S., & Antoniu, G. (2016). Fault tolerance in mapreduce: A survey. In *Resource management for big data platforms - algorithms, modelling, and high-performance computing techniques* (pp. 205–240).

Meng, X., Bradley, J. K., Yavuz, B., Sparks, E. R., Venkataraman, S., Liu, D., ... Talwalkar, A. (2016). Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, *17*, 34:1–34:7.

Naacke, H., Curé, O., & Amann, B. (2016). SPARQL query processing with apache spark. *CoRR*, *abs/1604.08903*.

Neumann, T., & Weikum, G. (2010). The RDF-3X engine for scalable management of RDF data. *VLDB J.*, *19*(1), 91–113.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., ... Zenger, M. (2004). *The scala language specification.*

Papakonstantinou, V., Flouris, G., Fundulaki, I., Stefanidis, K., & Roussakis, G. (2016). Versioning for linked data: Archiving systems and benchmarks. In *Proceedings of the workshop on benchmarking linked data (BLINK 2016) co-located with the 15th international semantic web conference (iswc), kobe, japan, october 18, 2016.*

Papakonstantinou, V., Flouris, G., Fundulaki, I., Stefanidis, K., & Roussakis, Y. (2017). Spbv: Benchmarking linked data archiving systems. In *Joint proceedings of BLINK2017: 2nd international workshop on benchmarking linked data and nliwod3: Natural language interfaces for the web of data co-located with 16th international semantic web conference (ISWC 2017), vienna, austria, october 21st - to - 22nd, 2017.*

Sande, M. V., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E., & de Walle, R. V. (2013). R&wbase: git for triples. In *Proceedings of the WWW2013 workshop on linked data on the web, rio de janeiro, brazil, 14 may, 2013.*

Schätzle, A., Przyjaciel-Zablocki, M., Berberich, T., & Lausen, G. (2015). S2X: graph-parallel querying of RDF with graphx. In *Biomedical data management and graph online querying - VLDB 2015 workshops, big-o(q) and dmah, waikoloa, hi, usa, august 31 - september 4, 2015, revised selected papers* (pp. 155–168).

Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., & Lausen, G. (2016). S2RDF: RDF querying with SPARQL on spark. *PVLDB*, *9*(10), 804–815.

Spahiu, I. B. (2017). *Profiling linked data* (Unpublished doctoral dissertation). BICOCCA. (unpublished thesis)

Stefanidis, K., Chrysakis, I., & Flouris, G. (2014). On designing archiving policies for evolving rdf datasets on the web. In E. Yu, G. Dobbie, M. Jarke, & S. Purao (Eds.), *Conceptual modeling: 33rd international conference, er 2014, atlanta, ga, usa, october 27-29, 2014. proceedings* (pp. 43–56). Cham: Springer International Publishing.

Tappolet, J., & Bernstein, A. (2009). Applied temporal RDF: efficient temporal

querying of RDF data with SPARQL. In *The semantic web: Research and applications, 6th european semantic web conference, ESWC 2009, heraklion, crete, greece, may 31-june 4, 2009, proceedings* (pp. 308–322).

Troullinou, G., Roussakis, G., Kondylakis, H., Stefanidis, K., & Flouris, G. (2016). Understanding ontology evolution beyond deltas. In *Proceedings of the workshops of the EDBT/ICDT 2016 joint conference, EDBT/ICDT workshops 2016, bordeaux, france, march 15, 2016.*

Udrea, O., Recupero, D. R., & Subrahmanian, V. S. (2010). Annotated RDF. *ACM Trans. Comput. Log.*, *11*(2), 10:1–10:41.

Völkel, M., & Groza, T. (2006, October). SemVersion: An RDF-based Ontology Versioning System. In M. B. Nunes (Ed.), *Proceedings of iadis international conference on www/internet (iadis 2006)* (p. 195-202). Murcia, Spain.

Wauer, M., Both, A., Schwinger, S., Nettling, M., & Erling, O. (2015). Integrating custom index extensions into virtuoso RDF store for e-commerce applications. In *Proceedings of the 11th international conference on semantic systems, SEMANTICS 2015, vienna, austria, september 15-17, 2015* (pp. 65–72).

Zablith, F., Antoniou, G., d'Aquin, M., Flouris, G., Motta, H. K. E., Plexousakis, D., & Sabou, M. (2015). Ontology evolution: a process-centric survey. *Knowledge Eng. Review*, *30*(1), 45–75.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., ... Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX symposium on networked systems design and implementation, NSDI 2012, san jose, ca, usa, april 25-27, 2012* (pp. 15–28).

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. In *2nd USENIX workshop on hot topics in cloud computing, hotcloud'10, boston, ma, usa, june 22, 2010.*

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., ... Stoica, I. (2016a). Apache spark: a unified engine for big data processing. *Commun. ACM*, *59*(11), 56–65. Retrieved from `http://doi.acm.org/10.1145/2934664` doi: 10.1145/2934664

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., ... Stoica, I. (2016b). Apache spark: a unified engine for big data processing. *Communications of the ACM*, *59*(11), 56–65.

Zimmermann, A., Lopes, N., Polleres, A., & Straccia, U. (2012). A general framework for representing, reasoning and querying with annotated semantic web

data. *J. Web Sem.*, *11*, 72–95.