Evaluation of RDF Archiving strategies with Spark

Meriem Laajimi[1], Afef Bahri[2], and Nadia Yacoubi Ayadi[3]

[1] High Institute of Management Tunis, Tunisia
laajimimeriem@yahoo.fr
[2] MIRACL Laboratory, University of Sfax Tunisia
afef.bahri@gmail.com
[3] RIADI Research Laboratory, ENSI, University of Manouba, 2010 Tunisia
nadia.yacoubi.ayadi@gmail.com

Abstract. Over the last decade, 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 an emergent need for efficient RDF data archiving systems. In fact, applications need to access to not only the actual version of a dataset but equally to the previous ones in order to query and track data over time. Querying RDF dataset archives involves performance and scalability. The proposed RDF archiving systems or benchmarks are built on top of existing RDF query processing engine. Nevertheless, efficiently processing a time-traversing query over Big RDF data archives is more challenging than processing the same query over an RDF datastore. We propose in this paper to use a distributed system, namely Apache Spark, in order to evaluate RDF archiving strategies. We propose and compare different query processing approaches with a detailed experimentation.

Keywords: RDF archives · Versioning queries · SPARQL · SPARK · SPARK SQL .

1 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. The Linked Open Data cloud [3], as published in 22 August 2017 illustrates the important number of published datasets and their possible interconnections (1,184 datasets having 15,993 links). LODstats, a project constantly monitoring statistics reports 2,973 RDF datasets that incorporate approximately 149 billion triples. As a consequence, an emerging interest on what we call archiving of RDF datasets [13, 5, 8] has emerged raising several challenges that need to be addressed. Moreover, the emergent need for efficient web data archiving leads to recently developed Benchmarking RDF archiving systems such as BEAR (BEnchmark of RDF ARchives) [5] and EvoGen [8]. 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[1] project is focusing the problem of Benchmarking Big Linked Data. A new Benchmark SPBv was developed with some preliminary experimental results [11]. 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 have been done to process RDF linked data with existing Big data processing infrastructure like Hadoop or Spark [9, 12]. 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. Many versioning strategies have been proposed in the literature: (a) Independent Copies (IC), (b) Change Based copies (CB) or Deltas and (c) Timestamp-based approaches (TB) [13, 5, 10]. 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. Using Big data processing framework would give advantage to the Independent Copies/Temporal approaches as CB approach may induce the computing of one or more versions on the fly.
Given the fact that we use IC approach and that all the versions are stored, querying evolving RDF datasets data represents the most important challenge beyond the use of RDF archiving system on top of Big data processing framework. Many types of RDF archives queries have been proposed: version materialization, delta materialization, single version and cross version query types. Which partitioning strategy would be adopted for treating theses queries. We note that in case of version Materialization, as all the version need to be loaded, the performance of the query processing does not depend on the used partitioning strategy. This is not the case of single/cross-time structured query where the use of partitioning may improve query performance [2, 9, 1].
In this paper, we use the in-memory cluster computing framework SPARK for managing and querying RDF data archive. The paper is organized as follows. Section 2 presents existing approaches for the design and evaluation of RDF archiving and versioning systems. Section 3 presents our approach for managing and querying RDF dataset archives with SPARK. A mapping of SPARQL into SPARK SQL and a discussion of the cost of versionning RDF queries are presented in section 4. Finally, an evaluation of RDF versioning queries is presented in section 5.

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

2 Related Works

Over the last decade, the published data is continuously growing leading to the explosion of the data on the Web and the associated Linked Open Data (LOD) in various domains. This evolution naturally happens without pre-defined policy hence the need to track data changes and thus the requirement to build their own infrastructures in order to preserve and query data over time. We 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.

Three versionning approaches are proposed 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) [10, 5]. We talk about hybrid approaches when the above techniques are combined [13]. The IC approach manages each version of a dataset as an isolated one while the CB approach stores only the changes that should be kept between versions also known as delta. The advantage beyond the use of IC or CB approaches depends on the ratio of changes occurring between consecutive versions. If only few changes are kept, CB approach reduces space overhead compared to the IC one. Nevertheless, if frequent changes are made between consecutive versions, IC approach becomes more storage-efficient than CB. Equally, the computation of full version on-the-fly with CB approach may cause overhead at query time. To resolve this issue, authors in [13] propose hybrid archiving policies to take advantage of both the IC and CB approaches. In fact, a cost model is conceived to determine what to materialize at a given time: a version or a delta.

Archiving systems not only need to store and provide access to different versions, but should also be able to provide query processing functionalities [13, 5, 10]. Four query types are mainly discussed in the literature. We note version materialization which is a basic query where a full version is retrieved. Delta materialization which is a type of query performed on two versions to detect changes occurring at a given moment. Single-version and cross-version queries correspond to queries, namely SPARQL, performed respectively on a single or different versions. The authors in [10] propose a taxonomy containing eight queries classified according to their types (materialization, single version, cross version) and focus (version or delta).

Moreover, the emergent need of efficient web data archiving leads to recently developed Benchmarking RDF archiving systems such as BEAR (BEnchmark of RDF ARchives) [5, 6] EvoGen [8] and SPBv [11]. 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 project is focusing on the problem of Benchmarking Big Linked Data. In this context, EvoGen is proposed as a configurable and adaptive data and query load generator. EvoGen extends the LUBM ontology and is configurable

in terms of archiving strategies and the number of versions or changes. Recently, new Benchmark SPBv was developed with some preliminary experimental results [11]. Similar to EvoGen, SPBv proposes a configurable and adaptive data and query load generator.
Concerning Big RDF dataset archives, the use of a partitioning strategy depends on the shape of the used SPARQL queries. Many works handles the Big RDF data by a simple hash partitioning on their RDF subject [2, 9] which improves the performance with star queries. For example, subject based partitioning strategy seems to be more adapted for treating Star based query shape (s,p,o) as all the triples for which the query is written are stored in the same node even though they do not belong to the same version which may accurate the performance of cross-version queries. For example, to follow the evolution of a given person career over time, we need to ask a star shape query of the form (?x,hasJob, ?y) on different versions.
Even though with simple query, the performance often drop significantly for queries with large diameter. The authors in [12] propose a novel approach to partition RDF data, named ExtVP (Extended Vertical Partitioning). In fact, based on pre-evaluation of the data, many RDF triple patterns are used to partition the data into partition tables (a partition for each triple pattern). That is, a triple query pattern can be retrieved by only accessing the partition table that bounds the query leading to a reduction of the execution time. The problem become more complex when we ask about cross-version join queries. For example we may need to know if the diploma of a person ?x has any equivalence in the RDF dataset archive: (?x hasDip ?y) on version V_1 and (?y hasEqui ?z) on versions $V_2, ..., V_n$. Realizing a partition on the subject for this kind of query may engender many transfer between nodes.

3 RDF dataset archiving on Apache Spark

In this section, we present the main features of Apache SPARK cluster computing framework we show how we can use it for change detection and RDF dataset versioning.

3.1 Apache Spark

Apache Spark [15] 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) [14] 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 [7] and Spark SQL, a Spark module for processing semi-structured data.
SPARK SQL [4] 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 [4]. SPARK SQL can automatically infer their schema and data types from the language type system.

3.2 RDF Dataset storage and change detection

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 or as a table in Hive or any external database system is mapped into a SPARK dataframes (equivalent to tables in a relational database) with columns corresponding respectively to the subject, property, object, named graph and eventually a tag of the corresponding version.
In order to obtain a view of a dataframe named "table", for example, we execute the following SPARK SQL query:

SELECT * FROM table

Figure 1 shows a view of a SPARK dataframe containing two versions of the RDF dataset defined in the example used in the paper.

IRI	Subject	Predicate	Object	Version
g1	toto	hasJob	analyst	V1
g2	mimi	hasJob	develop	V1
g1	toto	hasJob	dataSc	V2
g3	mimi	hasJob	develop	V2

Fig. 1. Example of a Dataframe with RDF dataset attributes.

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'

In order to define delta between versions we define the following SQL SPARK query:

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

3.3 RDF Dataset partitioning

In this section, we present the principle that we adopt for the partitioning of RDF dataset archives for efficiently executing single version and cross-versions queries (figure 2). Concerning version and delta materialization queries, all the data (version or delta) will be loaded and no partition is needed.

- First of all, we load RDF datasets in a N-triple format from HDFS as input.
- Then, a mapping is realized 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 and the query result is returned.

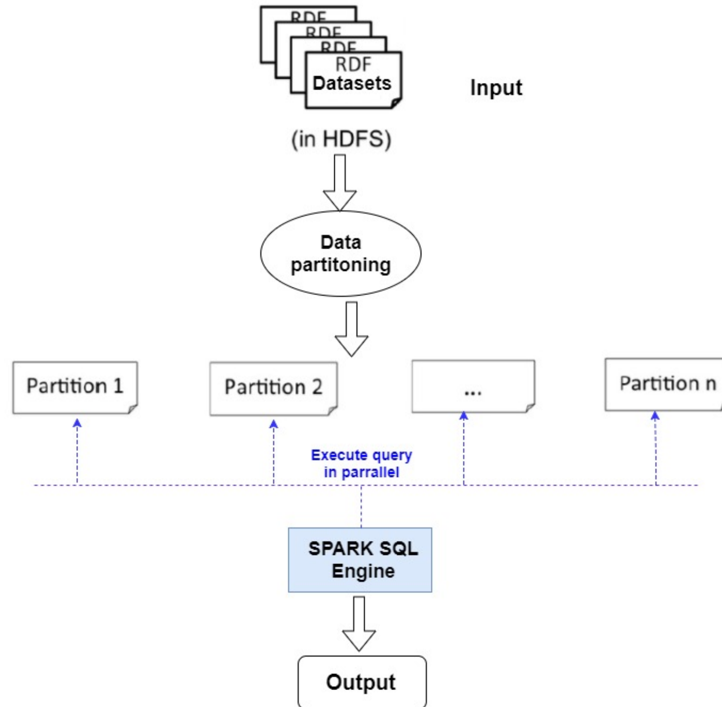

Fig. 2. Query execution with data partition of single version and cross-version queries.

4 Querying RDF dataset archives with SPARK SQL

In this section, we define basic RDF archiving queries (version/delta materialization, single/cross version query) with SPARK SQL.

4.1 Querying RDF dataset archives with SPARK SQL

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 ='Vi'
```

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

```
SELECT Subject,Predicate,Object FROM table WHERE Version='Vi'
MINUS
SELECT Subject,Predicate,Object FROM table WHERE Version='Vj'
UNION
SELECT Subject,Predicate,Object FROM table WHERE Version='Vj'
MINUS
SELECT Subject,Predicate,Object FROM table WHERE Version='Vi'
```

- **Single-version query**: $[[Q]]_{V_i}$. We suppose here a simple query Q which asks for all the subject in the RDF dataset.

```
SELECT Subject FROM table WHERE Version=Vi
```

- **Cross-version structured query**: $Join(Q_1, V_i, Q_2, V_j)$. 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:

$$\text{SELECT * FROM df}_i \\ \text{INNER JOIN df}_j \\ \text{ON df}_i\text{.Subject = df}_j\text{.Subject}$$

4.2 From SPARQL to SPARK SQL

SPARK SQL is used in [9, 12] 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. Theses mapping rules are used without considering SPARQL query shapes. 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:

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.

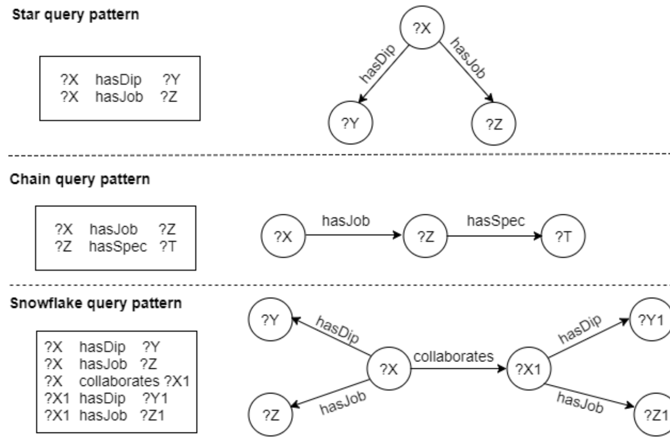

Fig. 3. SPARQL graph pattern shapes.

3. Snowflake pattern: this query pattern results from the combination of many star patterns connected by short paths.

When we query RDF dataset archives, we have to deal with SPARQL query shapes only in single version and cross-version queries. We propose in the following a mapping from SPARQL to SPARK SQL based on query shapes:

– Star pattern: a Star SPARQL query with n triple patterns P_i is mapped into a SPARK SQL query with n-1 joins on the subject attribute. If we consider a SPARQL query with two triple patterns P_1 and P_2 of the form (x_1?,p_1,?y_1) and (x_1?,p_2,?z_2), the dataframes df_1 and df_2 corresponding respectively to the query patterns P_1 and P_2 are defined with SPARK SQL as follows:

  df_1= "SELECT Subject, Object FROM table
  WHERE Predicate = 'p_1"'
  df_2= "SELECT Subject, Object FROM table
  WHERE Predicate = 'p_2"'

  For example, 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"'

  We give in the following the obtained SPARK SQL query:

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

- Chain pattern: a chain SPARQL query with n triple patterns t_i is mapped into a SPARK SQL query with n-1 joins object-subject (or subject-object):

$$join(join(join(join(df_1, df_2), df_3), df_4), ..., df_n)$$

If we consider a SPARQL query with two triple patterns P_1 and P_2 of the form (x_1?,p_1,?z_1) and (z_1?,p_2,?t_2), the dataframes df_1 and df_2 corresponding respectively to the query patterns P_1 and P_2 are defined with SPARK SQL as follows:

df_1= "SELECT Subject, Object FROM table
WHERE Predicate = 'p_1"'
df_2= "SELECT Subject, Object FROM table
WHERE Predicate = 'p_2"'

For example, given a SPARQL query 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

- 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 results obtained by executing Q on versions V_1,...,V_n respectively. The final result is obtained by realizing the union of the T_i. What we can conclude here is that the number of versions does not increase the number of joins which only depends on the shape of the query. Given a SPARQL query with a triple pattern P of the form (x_1?,p,?y_1) defined on

different versions V_1 and V_2, the SPARK SQL query is defined as follows:

SELECT Subject, Object FROM table
WHERE Predicate = 'p' and Version = 'V_1'
UNION
SELECT Subject, Object FROM table
WHERE Predicate = 'p' and Version = 'V_2'

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 a SPARQL patterns P_1 and P_2 of the form (x_1?,p_1,?z_1) and (z_1?,p_2,?t_2) defined on different versions V_1 and V_2, the dataframes df_1 and df_2 corresponding respectively to the query patterns P_1 and P_2 are defined with SPARK SQL as follows:

   df_1= "SELECT Subject, Object FROM table
   WHERE Predicate = 'p_1' and Version = 'V_1'
   df_2= "SELECT Subject, Object FROM table
   WHERE Predicate = 'p_2' Version = 'V_2'

   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

   Given df_1,...,df_n the different dataframes 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 df_i. In the worst case we may need to compute n-1 joins:

   $$join(join(join(join(df_1, df_2), df_3), df_4), ..., df_n)$$

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

5 Experimental evaluation

Evaluation was performed in cloud environment 'Amazon Web services' using EMR (Elastic Map reduce) as a platform. The data input files were saved on S3 Amazon. The experiments were done in a cluster with three nodes (one master and 2 core nodes) using m3.xlarge as an instance type. We use the BEAR dataset

Versions	Triples	Added triples	Deleted triples
Version 1	30,035,245	-	-
Version 5	27,377,065	6,922,375	9,598,805
version 10	28,910,781	9,752,568	11,092,386
version 15	33,253,221	14,110,358	11,150,069
version 20	35,161,469	18,233,113	13,164,710
version 25	31,510,558	16,901,310	15,493,857
version 30	44,025,238	30,697,869	16,797,313
version 35	32,606,132	19,210,291	16,645,753
version 40	32,923,367	18,125,524	15,312,146

Table 1. RDF dataset description

Benchmark which monitors more than 650 different domains across time and is composed of 58 snapshots. The description of the dataset is given in table 1. In the following we present the evaluation[2] of versioning queries on top of SPARK framework. The evaluation concerns four query types: version and delta materialization, single version and cross-version queries respectively.

5.1 Version and Delta Materialization

The content of the entire version (resp. Delta) is materialized. For each version, the average execution time of the queries was computed. Based on the plots shown in figure 4, we 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, versions in CB approach are not already stored and need to be computed each time we want to query a given version (resp. Delta).

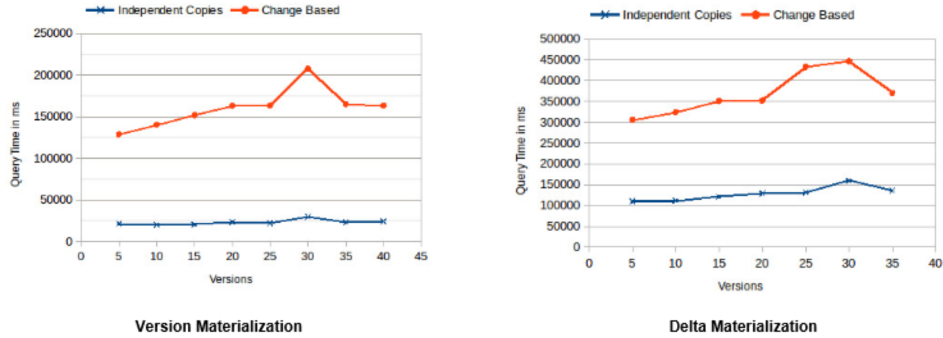

Fig. 4. Version and Delta Materialization IC and CB approaches

[2] https://github.com/meriemlaajimi/Archiving

5.2 Single-version queries

We realize different experimentations with subject, predicate and object based queries or a combination of them. Figure 5 concerns single version queries where the object and/or predicate is given whereas the subject corresponds to what we ask for. The analysis of the obtained plots shows that the use of partitioning ameliorates the query execution times. Nevertheless, using query with individual triple pattern does not need an important number of I/O operations. That is, the real advantage beyond the use of partitioning is not highlighted for this kind of queries which is not the case of cross-version queries.

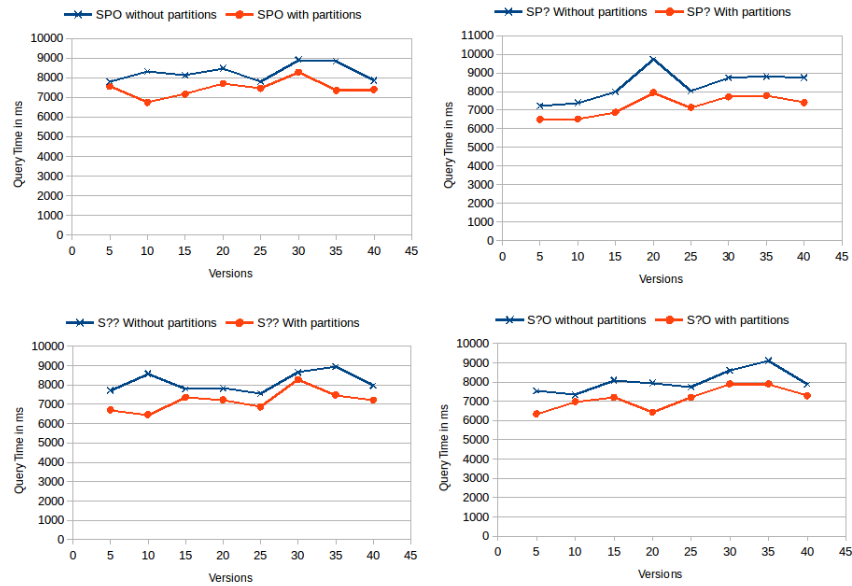

Fig. 5. Single version queries (Subject)

5.3 Cross-version queries

In this section, we focus on Cross-version queries. The first series of tests are realized with STAR query shape of the form (?X, p, ?Y) and (?X, q, ?Z). We give on the following an example of query used for experimentations. The obtained execution times are shown in table 3. We note that the advantage beyond the use of partitioning is highlighted for this kind of queries compared to the result obtained with single triple queries (subsection 5.2). As we can see in figure 6, the use of partitioning ameliorates execution times. In fact, Star query invokes triple patterns having the same subject value. When we use partitioning on subject

attribute, theses triples patterns are loaded in the same partition and no transfer is needed between nodes [9]. In our approach, RDF triple patterns belonging to different versions and having the same subject are equally loaded in the same partition.

Versions	Triples	SQ without partitions (ms)	SQ with partitions (ms)
V_1 and V_5	57,412,310	15005.226	12431.357
V_5 and V_{10}	56,287,846	15808.009	13531.05
V_{10} and V_{15}	62,164,002	16482.251	13223.434
V_{15} and V_{20}	68,414,690	16563.959	14165.733
V_{20} and V_{25}	66,672,027	15839.788	14532.462
V_{25} and V_{30}	75,535,796	16158.124	15053.127

Table 2. Query time evaluation of Star query(SQ)

We equally realize a second series of tests using Chain queries with two triples patterns of the form (?X, p, ?Y) and (?Y, q, ?Z). Table 3 shows the execution times obtained with the Chain query. As we can equally see in figure 6, the use of partitioning ameliorates execution times. We note that for executing Chain queries, object-subject (or subject-object) joins are needed and data transfer between nodes is necessary for executing queries. In fact, as the partition is realized on subject attribute, to realize the join between subject and object values we need to transfer the triples having such an object value from other partitions. After that, the execution times obtained with Chain query are superior to those obtained with Star query shapes.

Versions	Triples	CQ without partitions (ms)	CQ with partitions (ms)
V_1 and V_5	57,412,310	15002.811	13630.838
V_5 and V_{10}	56,287,846	16072.282	14029,593
V_{10} and V_{15}	62,164,002	16939.459	14395.548
V_{15} and V_{20}	68,414,690	17670.103	14247.463
V_{20} and V_{25}	66,672,027	16999.656	14681.513
V_{25} and V_{30}	75,535,796	19044.695	16257.424

Table 3. Runtime evaluation of Chain query (CQ)

What we can conclude is that, using partition with SPARK is favourable for executing cross-versions queries (Star queries). Nevertheless, Chain (or Snowflake) queries need to be deeply addressed as the number of join between non partitioned data may affect query execution times.

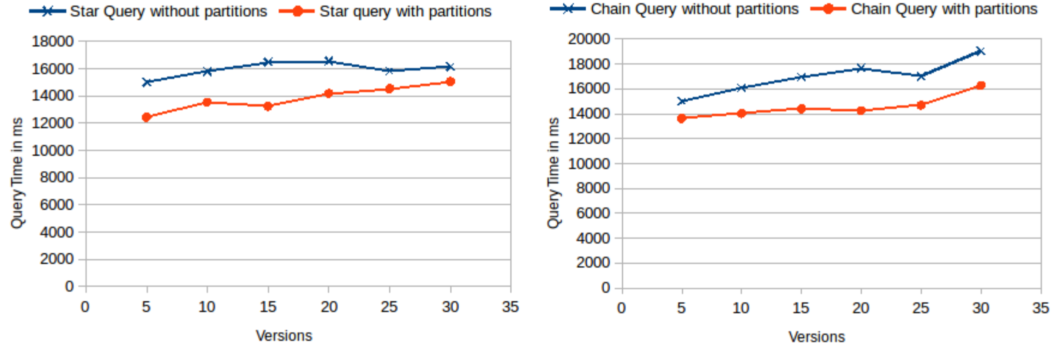

Fig. 6. Cross-version queries: Star and Chain query shapes

6 Conclusion

In this paper, 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 Copies approaches), the types of RDF archives queries, the size of versions, the shape of SPARQL queries and finally the data partitioning strategy. What we can conclude is that, using partitioning on the subject attribute with SPARK is favourable for executing cross-version Star queries as the execution of this query type does not need transfer between nodes which is not the case of cross-version Chain queries.
We note that, the number of patterns used in the Chain query as well as the number of versions have an implication on the number of join operations used to execute the query and by the way the number of data transfers. Different issues need to be considered, namely, which partitioning strategy will be adapted for efficiently executing cross-version Chain queries. In the future works we project to use different partitioning strategies [12] and to define execution plan of join operations [9] by taking into consideration, the size of a version, the number of versions and the shape of used queries.

References

1. Abdelaziz, I., Harbi, R., Khayyat, Z., Kalnis, P.: A survey and experimental comparison of distributed SPARQL engines for very large RDF data. PVLDB **10**(13), 2049–2060 (2017)
2. Ahn, J., Im, D., Eom, J., Zong, N., Kim, H.: G-diff: A grouping algorithm for RDF change detection on mapreduce. In: Semantic Technology - 4th Joint International Conference, JIST, Chiang Mai, Thailand, November 9-11, Revised Selected Papers. pp. 230–235 (2014)

3. Andrejs Abele, John P. McCrae, P.B.A.J., Cyganiak, R.: Linking Open Data cloud diagram 2018. http://lod-cloud.net// (2018), [Online; accessed April-2018]
4. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark SQL: relational data processing in spark. In: Proceedings of the SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4. pp. 1383–1394 (2015)
5. Fernández, J.D., Umbrich, J., Polleres, A., Knuth, M.: Evaluating query and storage strategies for rdf archives. In: Proceedings of the 12th International Conference on Semantic Systems. pp. 41–48. ACM, New York, NY, USA (2016)
6. Fernández, J.D., Umbrich, J., Polleres, A., Knuth, M.: Evaluating query and storage strategies for rdf archives. Semantic web journal IOS Press (2017)
7. Graube, M., Hensel, S., Urbas, L.: 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, Leipzig, Germany, September 2nd (2014)
8. Meimaris, M., Papastefanatos, G.: The evogen benchmark suite for evolving rdf data. In: MEPDaW Workshop, Extended Semantic Web Conference (2016)
9. Naacke, H., Curé, O., Amann, B.: SPARQL query processing with apache spark. CoRR (2016)
10. Papakonstantinou, V., Flouris, G., Fundulaki, I., Stefanidis, K., Roussakis, G.: Versioning for linked data: Archiving systems and benchmarks. In: Proceedings of the Workshop on Benchmarking Linked Data, Kobe, Japan, October 18 (2016)
11. Papakonstantinou, V., Flouris, G., Fundulaki, I., Stefanidis, K., Roussakis, Y.: Spbv: Benchmarking linked data archiving systems. In: 2nd International Workshop on Benchmarking Linked Data, ISWC
12. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on spark. PVLDB **9**(10), 804–815 (2016)
13. Stefanidis, K., Chrysakis, I., Flouris, G.: On Designing Archiving Policies for Evolving RDF Datasets on the Web, pp. 43–56 (2014)
14. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: 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, San Jose, CA, USA, April 25-27. pp. 15–28 (2012)
15. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)