Presto-RDF: SPARQL Querying over Big RDF Data

Mulugeta Mammo(✉) and Srividya K. Bansal(✉)

Arizona State University, Mesa, AZ, USA
{mmammo,srividya.bansal}@asu.edu

Abstract. There has been a rapid increase in the amount of Resource Description Framework (RDF) data on the web. The processing of large volumes of RDF data requires an efficient storage and query-processing engine that can scale well with the volume of data. In the past two and half years, however, heavy users of big data systems, like Facebook, noted limitations with the query performance of these big data systems and began to develop new distributed query engines for big data that do not rely on map-reduce. Facebook's Presto is one such example. This paper proposes an architecture based on Presto, called Presto-RDF, that can be used to process big RDF data. An evaluation of performance of Presto in processing big RDF data against Apache Hive is also presented. The results of the experiments show that Presto-RDF framework has a much higher performance than Apache Hive and native RDF store - 4Store and it can be used to process big RDF data.

Keywords: Database performance · Evaluation · Querying · Semantic web data

1 Introduction

Semantic Web is the web of data that provides a common framework and technologies for sharing data and reusing data in various applications and enterprises. Resource Description Framework (RDF) enables the representation of data as a set of linked statements, each of which consists of a subject, predicate, and object called a triple. RDF datasets, consisting of millions of triples, form a network of directed graph (DG) and are stored in systems called triple-stores. A query language standard, SPARQL, has also been developed to query RDF datasets. For the Semantic Web to work, both triple-stores and SPARQL query processing engines have to scale well with the size of data. This is especially true when the size of RDF data is too big such that it is difficult, if not impossible, for conventional triple-stores to work with. In the past few years, however, new advances have been made in the processing of large volumes of data sets, aka big data, which can be made to use for processing big RDF data. In this regard, various research studies that use big data technologies for RDF data processing have been published [1]–[3]. The initial attempts to address this issue focused on optimizing native RDF stores as well as conventional relational databases management systems. But as the volume of RDF data grew to exponential proportions, the limitations of these systems became apparent and researchers began to focus on using big data analysis tools, most notably Hadoop, to process RDF data. Various studies and benchmarks that evaluate these tools for RDF data processing have been published [1], [4]–[6].

In the past two and half-years, new trends in big data technology have emerged that use distributed in-memory query processing engines based on SQL syntax. Some of these tools include: Facebook Presto [7], Apache Shark, and Cloudera Impala. These tools promise to deliver high performance query execution than traditional Hadoop system like Hive. The motivation of this paper is to validate this claim for big RDF data – i.e. if these new in-memory query processing models work well to deliver faster response times for SPARQL queries, which must be translated to SQL. This paper investigates if there is a gain in query execution performance, compared to Hive and 4store, by storing big RDF data in HDFS and using in-memory processing engine instead of MapReduce. Specifically, it addresses the following questions:

- Is it feasible to store big RDF data in HDFS and get improved query execution time, compared to Hive and native RDF stores like 4store, by translating SPARQL queries into SQL and then using the Presto distributed SQL query processing engine to run the translated queries?
- How much improvement, in query response time, can be attained by using in-memory query processing engine, e.g. Presto, against native RDF stores, like 4store, and other query processing engines based on MapReduce, like Hive?
- How do different RDF storage schemes in HDFS affect the performance of SPARQL queries?
- Is it possible to construct an end-to-end distributed architecture to store and query RDF datasets?

The rest of the paper is organized as follows: Related work is presented in section 2. The architecture of Presto-RDF framework and RDF storage strategies are presented in section 3. Section 4 describes the experimental setup for performance evaluation of Presto-RDF and results. Section 5 presents conclusions and future work.

2 Related Work

This section presents a review of related works that propose and evaluate different distributed SPARQL query engines. It also presents a review of two systems, Apache Spark and Cloudera Impala, which are similar to Facebook Presto. Different RDF storage schemes are discussed in section 2.3.

2.1 Distributed SPARQL

A distributed SPARQL query engine based on Apache Jena ARQ has been proposed [11]. The query engine extends Jena ARQ and makes it distributed across a cluster of machines. The extension involves re-designing some parts of Jena ARQ. Document indexing and pre-computation joins were also used to optimize the design. The results of the experiments that were conducted showed that the distributed query engine scaled well with the size of RDF data but its overall performance was very poor. The query engine, unlike Facebook Presto, uses MapReduce. Scalable RDF stores have been proposed that efficiently perform distributed Merge and Sort-Merge [9].

Marcello Leida et al. [12] propose a query processing architecture that can be used to efficiently process RDF graphs that are distributed over a local data grid. The architecture has no single point of failure and no specialized nodes – which is a different than Hadoop. They propose a sophisticated non-memory query planning and execution algorithm based on streaming RDF triples. Presto uses a distributed in-memory query-processing algorithm.

Xin Wang et al. [13] discuss how the performance of a distributed SPARQL query processing can be optimized by applying methods from graph theory. The framework presented in this paper translates a SPARQL query into its equivalent SQL query, and hence the query optimization that is done by Presto is for the SQL query and not for the SPARQL query.

A distributed RDF query processing engine based on a message passing has been proposed [14]. The engine uses in-memory data structures to store indices for data blocks and dictionaries. Just like Presto, the query-processing engine avoids disk I/O operations. The authors experimented their design over several types of SPARQL queries and were able to get a significant performance gain (as compared to Hadoop). TriAD is a distributed RDF engine where communication is based on Message Passing Interface [10]. Researchers have also studied partitioning of SPARQL queries instead of RDF datasets for significant performance gain [8].

2.2 Apache Spark and Cloudera Impala

Apache Spark and Cloudera Impala are two open-sources systems that are very similar to Facebook Presto. Both Apache Spark and Cloudera Impala offer in-memory processing of queries over a cluster of machines. According to Apache, Apache Spark is a "fast and general engine for large-scale data processing". Spark uses advanced Directed Acyclic Graph (DAG) execution engine with cyclic data flow and in-memory processing to run programs up to 100 (for in-memory processing mode) or 10 times faster (for disk processing mode) than Hadoop MapReduce [8]. Cloudera Impala is an open-source massively parallel processing (MPP) engine for data stored in HDFS. Cloudera Impala is based on Cloudera's Distribution for Hadoop (CDH) and benefits from Hadoop's key features – scalability, flexibility, and fault tolerance. Cloudera Impala, just like Presto, uses Hive Metastore to store the metadata information of directories and files in HDFS.

2.3 RDF Triple Stores

RDF triples can be stored and accessed in Hadoop Distributed File System (HDFS) by creating a relational layer on top of HDFS that maps triples into relational schemas. Hive, for example, allows storing data in HDFS based on a relational schema that defined by the user. Though there are some discrepancies among researchers regarding the naming and classification of relational schemas for RDF data, most researchers classify these schemas in to three groups [1], [4], [5], [15]:

- Triple table – the entire RDF data is stored as a single table with three columns – subject, predicate and object. Each triple is stored as a row in this table.

- Property-table – triples are grouped together by predicate name. In this scheme, all triples with the same predicate are stored in a separate table. Some researchers call property tables vertical partitioning.
- Cluster-property tables – in this scheme triples are grouped into classes based on correlation and occurrence of predicates. A triple is stored in the table based on the predicate class it belongs to.

In this research study, we use Presto that is a distributed SQL query engine that runs on a cluster of machines controlled by a single coordinator with hundreds or thousands of worker nodes. Our literature review shows that there are no other studies done on Presto for semantic data processing. Presto is optimized for ad–hoc analysis and supports standard ANSI SQL, including complex queries, aggregation, joins, and window function [7]. The client sends SQL query using the Presto command line interface to the coordinator that would then parse, analyze and plan the query execution. The scheduler, component within the coordinator, connects together the execution pipeline and assigns and monitors work to worker nodes that are closer to the data. The client gets data from the output stage, one of the worker nodes, which in turn pulls data from the underlying stages. In this project we propose architecture for Presto to process big RDF data.

3 Presto-RDF

This section proposes architecture, called Presto–RDF, which can be used to store and query big RDF data using the Hadoop Distributed File System (HDFS) and Facebook Presto. It also presents RDF–Loader, one of the key components of the architecture, which is used to read, parse and store RDF triples.

3.1 Architecture

Presto–RDF consists of the following components: a command line interface (CLI), a SPARQL to SQL compiler (RQ2SQL), Facebook Presto, Hive Metastore, HDFS, and RDF–Loader. Figure 1 illustrates the different components of the architecture. RDF data that is extracted from the Semantic Web is parsed and loaded into HDFS using a custom–made RDF-loader, which will also store metadata information on Hive Thrift Server. When a user submits a SPARQL query over a command line interface, the query is processed by a custom–made SPARQL to SQL converter, RQ2SQL, that translates the SPARQL query into SQL which would then be submitted to Facebook Presto. Presto, using its Hive connector and Hive Thrift Server, runs the SQL against HDFS and returns the result back to the CLI.

3.2 RDF–Loader

The purpose of the RDF–Loader is to load, parse, and store RDF data in HDFS. RDF–Loader implements four different RDF storage schemes and creates external Hive tables whose metadata is stored in the Hive Thrift server. Before the

RDF–Loader is executed the raw RDF data to be first processed is loaded into HDFS using this command: *hadoop fs –put file hdfs–dir*
Once the raw RDF data is uploaded, RDF–Loader runs several MapReduce jobs and stores the output back into HDFS. The structure of the data is defined by the schema that can be specified by users of the system. In order for the RDF–Loader to run and process raw RDF, the following input parameters are required:

- *database* – is the name of the database that will be created.
- *target* – is the type of RDF storage structure, i.e. the type of schema. There are four options: *triples*, *vertical*, *wide*, and *horizontal*.
- *expand* – this option indicates if *qnames* are to be expanded.
- *server* – is the DNS name or IP address of the master node, NameNode, of the Hadoop cluster.
- *port* – is the port number Hadoop listens to connections.
- *input* – is the path of the HDFS directory that holds the raw RDF data.
- *output* – is the path of the HDFS directory the processed RDF data will be stored.
- *format* – defines the format of the output files as they are stored in HDFS. The current version of the Hive meta–store supports five different formats: SEQUENCEFILE, TEXTFILE, RCFILE, ORC, and AVRO. This study makes use of the TEXTFILE format.

The following sections discuss four different RDF storage strategies implemented by the RDF–Loader. The next section presents an analysis of performance of each of these storage strategies.

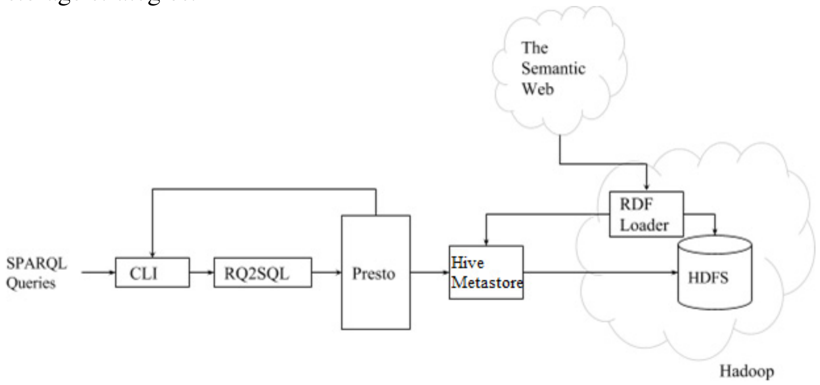

Fig. 1. Presto-RDF Architecture

3.3 Triple-Store Storage Strategy

In the triple-store storage scheme, an RDF triple is stored as is – resulting in a table with three columns: subject, predicate and object. If the raw RDF data has 30 million triples, the triple store strategy will have one table with 30 million rows.

The map–reduce algorithm that transforms the raw RDF data into the triples table is quite simple and shown in table 1.

```
map (String key, String value)
   // key: RDF file name
   // value: file contents
   for each triple in value
      emit_intermediate (subject + '\t ' + predicate, object)
reduce (String key, Iterator values)
   // key: subject and predicate delimited by tab
   // values: list of object values
   for each v in values
      emit (subject + '\t ' + predicate + object);
```

For an RDF dataset with n number of triples, the map algorithm has O(n) running time while the reducer, which is called once for each unique subject, has O(s*n) running time, where *s* is the number of unique subjects and *o* are the average number of object values per subject. Since, s*o=n, the total running time is O(n).

3.4 Wide–Table Storage Strategy

In the wide table RDF storage scheme, the raw RDF data is parsed and stored as a single table having one column for subject values, and multiple predicate columns for object values. The resulting table has the following schema:

WideTable (String subject, String predicate_1, String predicate_2, …, predicate_n)

Because it is unlikely that a subject has all the predicates found in the data set, this storage strategy will have a number of null values. For an RDF data set that has unique object values for a subject–predicate pair, this scheme would result in a table that has *s* number of rows, where *s* is the number of subjects in the data set. For example, given the following triple set in Table 1, the corresponding wide table representation for the triples would be as shown in Table 2.

Table 1. Example Triple set (a)

subject_1	predicate_1	object_1
subject_1	predicate_1	object_3
subject_2	predicate_2	object_3

Table 2. Wide table representation for example Triple set (a)

subject	predicate_1	predicate_2
subject_1	object_1	object_3
subject_2	null	object_3

If the dataset, however, contains multiple values for the same subject–predicate pair, the table will have multiple rows for the same subject. For example, given the following triple set in Table 3, the corresponding wide table representation for the triples would be as shown in Table 4.

Table 3. Example Triple set (b)

subject_1	**predicate_1**	**object_1**
subject_1	predicate_1	object_2
subject_2	predicate_2	object_3

Table 4. Wide table representation for example Triple set (b)

subject	predicate_1	predicate_2
subject_1	object_1	null
subject_1	object_2	null
subject_2	null	object_3

The algorithm for storing triples using the wide table storage scheme would get complicated if the data set contains subjects with multiple predicates (which is natural) and multiple object values for the same predicate. For example, given the following triple set in Table 5, the corresponding wide table representation for the triples would be as shown in Table 6.
The storage scheme, thus, forces new rows to be created for each unique subject–predicate pair. The map–reduce algorithm for the wide table storage scheme, as implemented in this study, is shown in the table below.

```
map (String key, String value)
  // key: RDF file name
  // value: file contents
  for each triple in value
    emit_intermediate (<subject, predicate>, <predicate, object>)
reduce (String key, Iterator values)
  // key: a <subject, predicate> pair
  // values: list of <predicate, object> pairs
  String subject = key.getSubject();
  String[] row = new String[1 + num_unique_predicates];
  int i = 0
  for each v in values
    row[i] = v.getObject();
    i++;
    emit (subject, row);
```

Table 5. Example Triple set (c)

subject_1	**predicate_1**	**object_1**
subject_1	predicate_1	object_2
subject_1	predicate_2	object_3
subject_2	predicate_2	object_4

Table 6. Wide table representation for example Triple set (c)

subject	**predicate_1**	**predicate_2**
subject_1	object_1	null
subject_1	object_2	null
subject_1	null	object_3
subject_2	null	object_4

3.5 Horizontal-Store Strategy

The horizontal storage scheme is similar to the wide table storage scheme in terms of the schema of the table. However, unlike the wide–table scheme, it optimizes the number of rows stored for subjects that have multiple object values for the same predicate. Given the example presented in the previous section in Table 5, the horizontal-store strategy stores the triples as shown in Table 7.

Table 7. Horizontal-store representation for example Triple set (c)

subject	**predicate_1**	**predicate_2**
subject_1	object_1	object_3
subject_1	object_2	null
subject_2	null	object_4

In this scheme, it is not necessary to create new rows for each unique subject–predicate pair. Instead, rows that are already created for the same subject, but for a different predicate will be used.

3.6 Vertical-Store Strategy

In the vertical storage scheme implemented in this research, the raw RDF data is partitioned into different tables based on the predicate values of the triples in the data with each table having two columns – the subject and object values of the triple. Thus, if the raw RDF data has 30 million triples that have 20 unique predicates, the vertical storage scheme will create 20 tables and stores the subject and object values of triples that share the same predicate in the same table. The map–reduce algorithm works with predicate as a key value and a pair of subject and object values as value:

```
map (String key, String value)
   // key: RDF file name
   // value: file contents
   for each triple in value
      emit_intermediate (predicate, <subject, object>);
reduce (String key, Iterator values)
   // key: predicate
   // values: list of <subject, predicate> pairs
   String table = key.replace_unwanted('_');
   MultipleOutputs<String, String> mos;
   for each v in values
      // create a directory table
     // write the subject, values inside the directory
     mos.write (v.getFirst(), v.getSecond(), table);
```

Because predicate values are URIs that contain non–alpha numeric characters, e.g. http://www.w3.org/1999/02/22–rdf–syntax–ns#, which cannot be used in naming directories, the reducer has to replace these characters with some other character, for example the underscore character, and creates the directory (which is considered as a table for the Hive Metastore). In the vertical storage scheme, for a raw RDF data that contains n number of triples, the mapper runs at $O(n)$ while the reducer runs at O($p*x$) where p and s are the number of unique predicates and subjects in the data set, respectively. In the worst case scenario, where there are as many unique predicates and subjects, the number of triples, the map-reduce algorithm for the vertical storage scheme runs at $O(n^2)$.

4 Benchmarking Presto-RDF

This section presents the experiments and the results conducted to benchmark the performance of Presto-RDF against Hive. A comparative measurement was also done on 4store – a native RDF store. Overall, two experimental setups were constructed for benchmarking the performance of Presto-RDF. The first setup was a 4-node cluster virtualized on a single 16GB memory machine. The second setup was 8-node cluster virtualized on the Windows Azure platform. The second setup was required because the experiments conducted used up the hard disk space and it was not possible to run queries on triples of more than 4 million. For the experiment, four benchmark queries from SP^{2}Bench [16], [17] were used and three different RDF storage schemes were evaluated – triple, vertical and horizontal stores. SP^{2}Bench is a SPARQL benchmark that is designed to test SPARQL queries over RDF triples stores as well as SPARQL–to–SQL re–write systems. SP^{2}Bench focuses on how well an RDF store supports the different SPARQL operators and their combination – known as operator constellations [16], [17]. SP^{2}Bench data model is based on the DBLP, http://www.informatik.uni–trier.de/~ley/db/, a computer science bibliography created in the 1980s and currently featuring more than 2.3 million articles. The SP^{2}Bench data generator can generate any number of triples based on what a user specifies. For the experiments conducted in this study, for example, triples of size 10, 20, and 30 million were generated.

4.1 Benchmarking Presto-RDF using 10, 20, and 30M Triples

The experiment was based on running four benchmark queries, from SP^{2}Bench with different degrees of complexity – query 1, 6, 8, and 11. The SP^{2}Bench use case is based on the DBLP. The experimental setup that was conducted involved setting up four and eight node clusters on Microsoft Windows Azure Platform. Each node in the cluster had a 2-core x86-64 processor, 14GB of memory, and 1TB of hard disk. Measurements were conducted for the four benchmark queries for 10, 20, and 30 million triples.

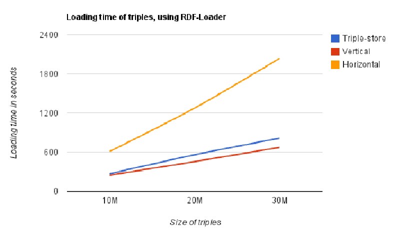

Fig. 2. Time Taken by RDF-Loader to Parse and Structure Raw RDF data

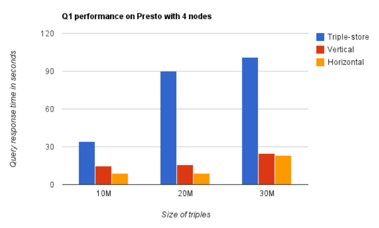

Fig. 3. Presto-RDF Query Processing Time of Ql over a 4 node Cluster

Loading Time

Once the RDF dataset is copied into HDFS, the RDF-Loader will parse and run a map-reduce job to convert the raw dataset to a structured dataset based on three storage schemas – triple-store, vertical and horizontal. The results of the measurement are shown in Figure 2. The performance of the RDF-loader has a linear relationship with the size of the triples. The horizontal store map-reduce algorithm always took much longer time than the triple-store and vertical store schemes. The result of running the above queries on Presto for a 4-node and 8-node cluster setup are shown in the figures below. For Q1, the vertical and horizontal stores have a much better performance than the triple-store schemas shown in Figure 3. This can be explained by looking into the SQL translations of the vertical and horizontal storage schemes – which have lesser rows involved in JOINs. This fact remains true when the number of nodes is increased from 4 to 8 – Figure 4. For Q1, increasing number of nodes resulted in performance improvement for the three storage schemes. Figure 5 below depicts the same.

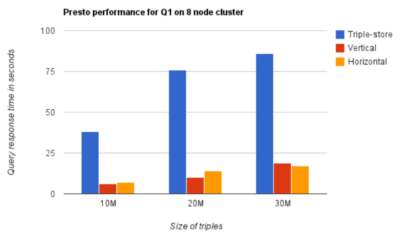

Fig. 4. Presto-RDF Query Processing Time of Ql over 8 Node Cluster

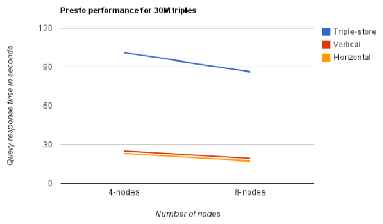

Fig. 5. Performance of Presto, with Increase in Number of Nodes, for Q1

Presto vs. Hive for Q1

Compared to Hive, Presto once again has a much higher performance. Figure 6 shows a comparison of Presto and Hive for 30M triples.

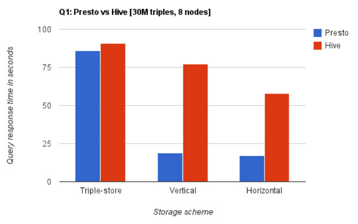

Fig. 6. Q1 Performance, for 30M Triples on 8 node Cluster

Evaluation Result for Q6

The SQL translations for Q6, unlike Q1, involve multiple JOINs for each of the three storage. The results of the evaluation on a 4-node and 8-node cluster are shown in Figure 7 and 8 below. The results of the evaluation above (Figure 7 and 8) indicate that the performance increased with increase in the number of nodes – see Figure 9 below. The vertical store, again, has a much better performance than the triple-store and horizontal store. Unlike Q1, however, where the horizontal store had a slightly better performance than the triple-store, the triple-store in Q6 had a slightly better performance than the horizontal store, especially as the size of the triples increases. This result can be explained by the fact that the horizontal store SQL for Q6, unlike the triple-store, involves multiple selections before making JOINs.

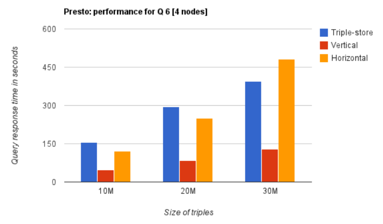

Fig. 7. Q6 Performance on Presto-RDF with 4 Nodes

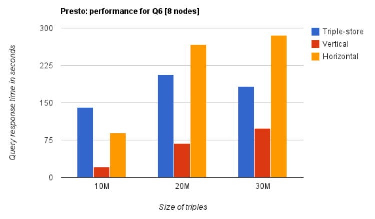

Fig. 8. Q6 Performance on Presto-RDF with 8 Nodes

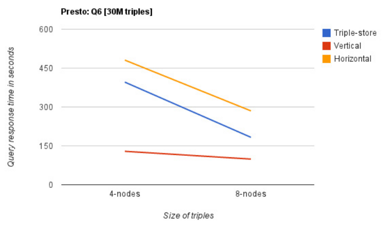

Fig. 9. Effect of Node Increase on Presto-RDF, for Q6 with 30M Triples

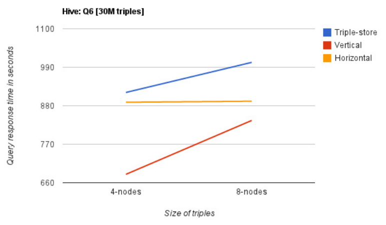

Fig. 10. Effect of Node Increase on Hive, for Q6 with 30M Triples

For Hive, unlike Presto-RDF, as the number of nodes was increased there was a drop in performance – which can be attributed to increase in replication across nodes and disk I/O operations – see figure 10.

Presto vs. Hive for Q6 and Q8

For Q6 as well, Presto-RDF has a much higher performance than Hive Evaluation. The SQL translations for Q8 involve multiple JOINs (just as the case were in Q6) and a UNION. The results have the same behavior as Q6 – the vertical store has a much better performance than the triple-store and horizontal stores, and Presto-RDF has a much higher performance than Hive. Figure 13 below shows the results of running the above queries over 10, 20 and 30M triples.

Because Q11 involves just one table that has less number of rows for the vertical and horizontal storage schemes than the triple-store (which is one table), the results shown above are expected. For 8 nodes, there is a performance improvement – see Figure 14.

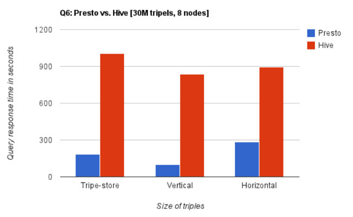

Fig. 11. Performance Comparison of Presto-RDF and Hive, for 30M Triples on 8-node Cluster

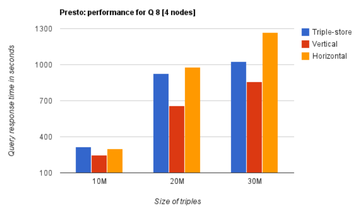

Fig. 12. Q8 performance, on Presto-RDF with 4 nodes

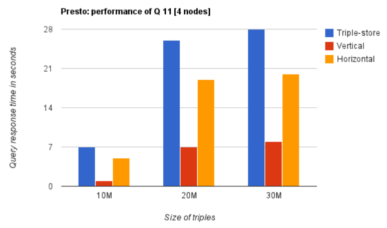

Fig. 13. Q11 Performance, on Presto-RDF with 4 Nodes

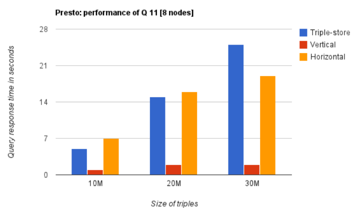

Fig. 14. Q11 Performance, on Presto-RDF with 8 Nodes

5 Conclusions and Future Work

This paper proposed a Presto-based architecture, Presto-RDF that can be used to store and process big RDF data. This paper also presented a comparative analysis of big RDF data using Presto, which uses in-memory query processing engine, and Hive, which uses Map Reduce to evaluate SQL queries. From the experiments conducted, following conclusions can be drawn:

- 4store has a much higher performance than Presto and Hive for small data sets. For bigger data sets (10M, 20M and 30M triples), however, 4store was simply unable to process the data and crashed. This is true when Presto, Hive and 4store are all tested with single-node setups.
- For all queries, Presto-RDF has a much higher performance than Hive.
- The vertical storage scheme has a consistent performance advantage than both the triple-store or horizontal storage schemes.
- As the size of data increases, the horizontal storage scheme performed relatively better than the triple-store scheme. This is unlike the articles reviewed during this research study, which ignore the horizontal scheme as being not efficient (because it has many null values).
- Increasing the number of nodes improved query performance in Presto but not in Hive. This can be explained by the fact that Hive replicates data across clusters and does IO operations – which increase as the size of nodes increase.

There are a number of areas to extend this study: this paper used a single benchmark, SP^{2}Bench. This work can be investigated on different benchmarks such as LUBM [18], BSBM [19], and DBPedia [6]. There are different optimization techniques that can be applied to the three storage schemas as well as to the RDF data directly. The RDF data is stored as a text file, which is not optimal. This work can be extended to test using RCFILE, ORC, and AVRO formats, which are better optimized than text file.

References

1. Luo, Y., Picalausa, F., Fletcher, G.H., Hidders, J., Vansummeren, S.: Storing and indexing massive RDF datasets. In: Semantic Search Over the Web, pp. 31–60. Springer (2012)
2. Cudré-Mauroux, P., Enchev, I., Fundatureanu, S., Groth, P., Haque, A., Harth, A., Keppmann, F.L., Miranker, D., Sequeda, J.F., Wylot, M.: NoSql databases for rdf: an empirical evaluation. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) ISWC 2013, Part II. LNCS, vol. 8219, pp. 310–325. Springer, Heidelberg (2013)
3. RDF, S.: Efficient RDF Storage and Retrieval in Jena2 (2003)
4. Sakr, S., Al-Naymat, G.: Relational processing of RDF queries: a survey. ACM SIGMOD Record **38**(4), 23–28 (2010)
5. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: Proc. of the Intl. Conf. on Very Large Data Bases, pp. 411–422 (2007)
6. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.-C.: DBpedia SPARQL benchmark – performance assessment with real queries on real data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 454–469. Springer, Heidelberg (2011)
7. Presto: Interacting with petabytes of data at Facebook. https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/1015178619 7628920. (accessed: December 02, 2014)
8. Hammoud, M., etal.: DREAM: distributed RDF engine with adaptive query planner and minimal communication. In: Proc. of Intl. Conf. on Vary Large Databases (VLDB 2015)
9. Papailiou, N., Tsoumakos, D., Konstantinou, I., Karras, P., Koziris, N.: H2RDF+: an efficient data management system for big RDF graphs. In: Proceedings of SIGMOD Conference, pp. 909-912 (2014)
10. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In: Proceedings of SIGMOD Conference, pp. 289-300 (2014)
11. Kulkarni, P.: Distributed SPARQL query engine using MapReduce. In: Master of Science, Computer Science, School of Informatics, University of Edinburgh (2010)
12. Leida, M., Chu, A.: Distributed SPARQL query answering over RDF data streams. In: 2013 IEEE International Congress on Big Data (BigData Congress), pp. 369–378 (2013)
13. Wang, X., Tiropanis, T., Davis, H.C.: Evaluating graph traversal algorithms for distributed SPARQL query optimization. In: Pan, J.Z., Chen, H., Kim, H.-G., Li, J., Wu, Z., Horrocks, I., Mizoguchi, R., Wu, Z. (eds.) JIST 2011. LNCS, vol. 7185, pp. 210–225. Springer, Heidelberg (2012)
14. Dutta, A.K., Theobald, M., Schenkel, R.: A Distributed In-Memory SPARQL Query Processor based on Message Passing (2012)
15. Harth, A., Hose, K., Schenkel, R.: Linked Data Management. In: CRC Press (2014)
16. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^ 2Bench: a SPARQL performance benchmark. In: Data Engineering, ICDE 2009, pp. 222–233 (2009)
17. The SP2Bench SPARQL Performance Benchmark. http://dbis.informatik.uni-freiburg.de/forschung/projekte/SP2B/. (accessed: December 02, 2014)
18. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Web Semantics: Science, Services & Agents on WWW **3**(2), 158–182 (2005)
19. Berlin SPARQL Benchmark. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlin sparqlbenchmark/. (accessed: December 02, 2014)