Efficient Distributed SPARQL Queries on Apache Spark

Saleh Albahli
College of Computer
Qassim University, Saudi Arabia

***Abstract*—RDF is a widely-accepted framework for describing metadata in the web due to its simplicity and universal graph-like data model. Owing to the abundance of RDF data, existing query techniques are rendered unsuitable. To this direction, we adopt the processing power of Apache Spark to load and query a large dataset much more quickly than classical approaches. In this paper, we have designed experiments to evaluate the performance of several queries ranging from single attribute selection to selection, filtering and sorting multiple attributes in the dataset. We further experimented with the performance of queries using distributed SPARQL query on Apache Spark GraphX and studied different stages involved in this pipeline. The execution of distributed SPARQL query on Apache Spark GraphX helped us study its performance and gave insights into which stages of the pipeline can be improved. The query pipeline comprised of Graph loading, Basic Graph Pattern and Result calculating. Our goal is to minimize the time during graph loading stage in order to improve overall performance and cut the costs of data loading.**

Keywords*—Semantic web; RDF; SPARQL; SPARK; GraphX; triple patterns

I. Introduction

The semantic web research came a long way from labelling different web pages and linking information to developing better processing systems and efficiently querying semantic web data for information. Today, the semantic web's methods, scale and its representational language are changing drastically. The web data is heterogeneous, it varies in size, semantics and quality of the content which can be true or not. The semantic web data is both high in volume and in velocity due to the wide adoption of digital medium. This has given rise to a lot of research questions. For example, how can we understand data patterns and integrate diverse data for faster access and better quality [1]. The current focus is not to develop new systems from scratch but to improve on existing frameworks with the use of new and improved technologies, such as MapReduce, Apache Spark, big data management, etc.

Resource Description Framework (RDF) is a schema-free data model [2] for linking and storing massive amount of both structured and semi-structured web data in a form of a graph *(subject predicate object)* where *subject* is linked to *object* by *predicate*. With the use of Uniform Resource Identifiers (URIs), RDF links and shares data in a directed and labelled graph where the edges represent the named link and the two nodes represent the two resources or endpoints. This allows data merging even if the underlying schemas are different. Unlike relational or hierarchical data models, RDF stores data in a data graph where there is no concept of roots or hierarchy. It just consists of resources with no single resource having any particular importance over another resource. It perfectly shows the relationship between different resources. Hence, the semantic web is a big global data graph defined in RDF with semantics embeded via RDFS (RDF Schema) [3]. More specifically, semantics triples are added to RDF data using RDFS by applying a set of inference rules to entail new facts, which are not explicitly asserted.

For querying RDF data, SPARQL Protocol and RDF Query Language (SPARQL) is used. SPARQL is currently the standard query language for retrieving and manipulating the data stored in RDF format. SPARQL provide a full range of analytical query operations such as JOIN, SORT and AGGREGATE without requiring a separate schema [4]. There are both programming languages and tools available with which a SPARQL query can be constructed. SPARQL allows to construct queries for RDF data as a set of subject-predicate-object triple. In a relational database terms, it can be considered as a table of three columns i.e. the subject column, the predicate column and the object column. Many RDF data processing systems rely on existing cluster computing engines for parallelized data processing. The current trend of big data needs fast loading and storage strategies to shorten the data ingestion time before query and faster results after querying [5]. Due to this, complex SPARQL queries over large RDF graphs generally have to combine a lot of distributed pieces of data through join operations.

The Apache Jena framework that is widely used as an open source Semantic Web Framework in Java for RDF and provides APIs to extract data from and write to data graph. On the other side, Apache Spark as a MapReduce framework proposes parallel computation using distributed main-memory data abstraction i.e. 1) Resilient Distributed Data Sets (RDD), a distributed lineage supported fault tolerant data abstraction for in memory computations and 2) Data Frames (DF), a compressed and schema-enabled data abstraction [6]. These data abstractions make programming queries easier by enabling translation and processing of high level query expressions such as SPARQL. On top of data abstraction, Spark provide data access models such as GraphX for processing semi-structured data and SPARQL queries over RDF data.

GraphX from Apache Spark is a component for graphs and graph-parallel computation. It reuses Spark's RDD concept, simplifies graph analytics tasks and makes operations on a directed multi-graph. It provides APIs for fast and robust development of a range of algorithms derived from graph theory and applied to search engines and social networks.

Why do we need to improve the query processing of RDF

datasets? As mentioned earlier, RDF data has increased in volume and variety available from many different sources on a range of topics. The RDF datasets such as Billion Triples Challenge datasets which are collected by web crawlers, the Linking Open Data Project and the recent conversion of the data.gov dataset are all examples RDF. With the growing availability of huge amount of RDF data, we need efficient ways of querying this data to extract the useful information in a reliable and fast manner [7]. Moreover, the goal is to support different data mining tasks while improving semantic web and exploring vast datasets for innovative insights.

We are achieving this goal by utilizing a cluster's parallelism i.e. our system is able to load and query a large dataset much more quickly than traditional approaches. We have designed experiments to evaluate the performance of several queries ranging from single attribute selection to selection, filtering and sorting multiple attributes in the dataset. We further experimented with the performance of queries using distributed SPARQL query on Apache Spark GraphX and studied different stages involved in this pipeline. We realized that minimizing the data loading time would significantly cut the cost and enhance query performance.

Motivation: RDF is a graph-like representation of knowledge. In this paper, we thus motivate the benefits of Apache Spark's framework GraphX to execute queries on RDF graph data with in memory processing ability of Spark. Our work is adopted the distributed manner and compare it with linear SPARQL query processing with different complexities, readability of queries and scalability using DBpedia dataset.

The importance of this study is that our approach performs better query than traditional approaches even with large datasets. Therefore, our approach of using SPARQL query processing is enhanced with Apache Spark GraphX on different sets of queries using large semantic web datasets. The execution of distributed SPARQL query on Apache Spark GraphX helped us study its performance and gave insights into which stages of the pipeline can be improved. The query pipeline comprised of Graph loading, Basic Graph Pattern and Result calculating. Our goal was to minimize the time during graph loading stage in order to improve overall performance and cut the costs of data loading.

II. Related Study

There are vast amount of research work available in semantic web, rdf, scalable pattern processing and much more. In this section, we will be giving overview of different research papers whose work is relevant to our work.

Chawla et al. [8] processed SPARQL queries on in-memory cluster computing engine Apache Spark and compared SPARQL execution strategies on different query shapes and data sets. They performed experiments on both real-world and synthetic data sets and proposed two new approaches for RDF data model. They were able to achieve a performance improvement by a factor of up to 2.4 on query execution time. The researchers concluded that hybrid query plans combining partitioned and broadcast joins improve query performance. Moreover, using DF representation when RDD exhausted the main memory of the cluster, helped store 10 times more data on the same cluster size with only small loss in performance.

Weaver et al. [6] study the usage of two distributed join algorithms, 1) partitioned join, and 2) broadcast join for the evaluation of basic graph patterns (BGP) using Apache Spark. They suggest through experimentation that hybrid join plans gives more flexibility and achieve better performance than single join plans.

Schätzle [9] aimed to optimize SPARQL queries in order to reduce their execution time. For this purpose, they have modified the conventional All Pair Shortest Path (APSP) algorithms which takes precomputed join costs between triples patterns in a SPARQL query graph using heuristic techniques. Finally, the authors have compared the Floyd Warshall and Johnson algorithms concluding the the former works faster in computing query plans.

Agathangelos et al. [10] devised a novel relational database to efficiently minimize query input size regardless of its pattern shape and diameter. The prototype system called S2RDF is developed on top of Spark and uses relational schema termed as ExtVP (Extended Vertical Partitioning) to execute SPARQL queries. It achieves sub-second runtimes for majority of queries on a billion triples RDF graph.

Naacke et al. [7] developed an application of parallel hash-joins for basic graph pattern matching without the need of any pre-processing, loading or global indexing of the RDF data. The approach relies on the cluster's (using 1024 processors) high bandwidth and fast memory to load and query data in parallel and close to real-time.

Auer et al. [11] discuss existing work in query processing of RDF data using Apache Spark. The RDF data model and the Spark APIs impact the implementation of the RDF quey processing approaches. RDDs have more flexibility for storage and partitioning and GraphX supports graph-parallel and data-parallel data processing.

Table I summarizes the various work conducted in RDF query processing giving details on methodology and findings. In this table, we clearly mention the reference of research work, Apache tool that was used, type of dataset, methodology adopted and finally the findings of each work.

III. Experimental Analysis

We evaluated our proposed model with real world use cases. Following are the details of the evaluation procedure.

A. Experimental Setup

We used two SPARQL implementations to compare and evaluate the performance of semantic queries over RDF datasets. For both of the scenarios we used Google Cloud infrastructure to setup the environment. All of our implementations were written in Java and Scala for performance and extensibility purposes.

1) Linear SPARQL: To establish a baseline for our experiments, we setup Apache Jena on a single node having 7.5 GB of memory, 2 virtual CPUs and 20 GB of persistent storage. Apache Jena is an opensource semantic web framework written in Java and providing APIs to extract data from and write to RDF graphs. Data is first loaded into an abstract model which is then used to query data using SPARQL query language.

TABLE I. COMPARISON TABLE OF PREVIOUS METHODOLOGIES FOR TRIPLE AND GRAPH MODEL.

Reference	Apache Spark Abstraction	Triple/Graph	Methodology	Finding
Naacke et al. (2016) [8]	Apache Spark SPARQL	Triple	Compared five SPARQL query processing strategies over an in-memory based cluster computing engine (Apache Spark) using different query shapes and datasets.	Hybrid query plans combining partitioned join and broadcast joins improve query performance in almost all cases. Moreover, SPARQL Hybrid RDD is bit more efficient than the hybrid DF solution due to the absence of a data compression/decompression overload. We can switch to DF representation if size of the RDDs saturates the main-memory of the cluster.
Naacke et al. (2017) [6]	Apache Spark SPARQL	Graph	Implemented and evaluated four SPARQL query processing strategies over different benchmark queries and data sets	The hybrid query plans combining partitioned and broadcast joins improved query performance in almost all cases and it naturally fits into the recent Spark-based S2RDF system to improve its performance.
Chawla et al. (2017) [9]	Apache Spark SPARQL	Graph	Modified the conventional All Pair Shortest Path (APSP) algorithms which take as input a pre-computed cost matrix of a graph-based SPARQL query.	Optimised SPARQL queries in order to reduce their execution time, APSP's Floyd Warshall algorithm worked faster in computing the plans than the Johnson algorithm..
Schatzle et al. (2016) [10]	Apache Spark SPARQL	Triple / GraphX	A relational partitioning schema for RDF data called ExtVP that uses a semi-join based preprocessing.	It efficiently minimizes query input size regardless of its pattern shape and diameter.
Agathangelos et al. (2018) [11]	Apache Spark, Spark SQL, GraphX, Graph-Frames	Triple / GraphX	Discussion on existing works with efficient query answering and novel ideas for improving query processing by exploiting data parallelization	Data partitioning is a key element for efficient query processing. Graph partitioning focuses on minimizing the edge-cut between partitions. GraphX has not been exploited yet towards this direction and could be an option to build such algorithms.

2) Distributed SPARQL: Distributing SPARQL queries over a cluster of commodity nodes not only improves system performance but will soon become essential as semantic web grows resulting and larger datasets to query and work with.

Apache Spark is one such distributed framework which supports in-memory iterative processing. While vanilla Spark is designed for general purpose distributed processing, it provides a number of libraries which can be used for a wide array of applications. A notable example is GraphX which provides with APIs for working with distributed graphs of nodes and arcs. Having properties pertaining to semantic data processing, GraphX is a perfect candidate to map an RDF graph and perform SPARQL queries in a distributed fashion. We used an open- source implementation S2X; SPARQL query processor for MapReduce based on Spark GraphX.

We used a cluster of 5 nodes to setup Apache Spark in Google Cloud. Each node had 3 GB of memory, 2 virtual CPUs and 20 GB of persistent storage. We used the latest Spark version at the time of writing i.e. Apache Spark version 2.4.3.

3) Dataset: The experiments were designed to ingest RDF datasets in N-Triples format. For datasets which were not in N-Triples format, a preprocessing step was performed to parse and convert the data to N-Triple format. All of the data was uploaded to Google Storage via *gsutil* i.e. google storage API for accessing data on distributed file system.

For smaller dataset, we created a few nodes graph manually. For larger dataset, we used DBPedia dataset [12]. As noted earlier, DBPedia dataset was originally in Turtle format which was preprocessed into N-Triples format before experiments.

B. Experiment Design

We designed several queries to evaluate the performance on available datasets. These ranged from simplest query on single attribute selection to selection, filtering and sorting on multiple attributes in the dataset.

The simplest query was to select person names in the dataset.

```
// Simple SPARQL query
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {?person foaf:name ?name .}
```

The second and comparatively complex query was on DBPedia dataset.

```
// Complex SPARQL query
SELECT *
WHERE {
?person
    <http://dbpedia.org/ontology/deathDate>
?deathDate .
?person <http://xmlns.com/foaf/0.1/page>
    ?page .
FILTER(?deathDate >= "1941-01-01"^^xsd:date)
FILTER(?deathDate <= "1942-01-01"^^xsd:date)
} order by ?deathDate;
```

Another query on DBPedia dataset was:

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX res: <http://dbpedia.org/resource/>
PREFIX rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:
<http://www.w3.org/2000/01/rdf-schema#>
SELECT COUNT(DISTINCT ?uri)
WHERE {
?uri rdf:type dbo:Film .
?uri dbo:starring res:Leonardo_DiCaprio .
}
```

Next we studied the execution of distributed SPARQL query on Apache Spark GraphX in terms of performance. This gave us insights into which stages can be improved further explained in Section III-D

C. Distributed SPARQL Performance

Fig. 1 shows the performance comparison results for linear SPARQL query processing and its distributed variant. As we can see, for smaller datasets, Linear query processing outperforms distributed processing. This happens mainly because large scale distributed systems do have overheads associated with system initialization and communication costs which only gets mitigated by processing enough data. This also enforces the fact that semantic processing needs to embrace parallel and distributed processing principals as the data to process keeps on increasing.

During the experiments, we did note that as the dataset increased, Apache Jena kept getting out of memory. We were able to handle the situation by increasing JVM's heap size. However, this is a temporary solution and there is a limit to size of heap on a single node. This again shows that single node processing for SPARQL queries on semantic data is by no means optimal.

D. Distributed SPARQL Analysis

To further study Spark based distributed SPARQL query processing, we analyzed different stages involved in the distributed SPARQL pipeline. Fig. 2 shows the percentage time spent in different stages while executing SPARQL query in a distributed environment using Apache Spark's GraphX library. In total, the query pipeline consists of three separate stages:

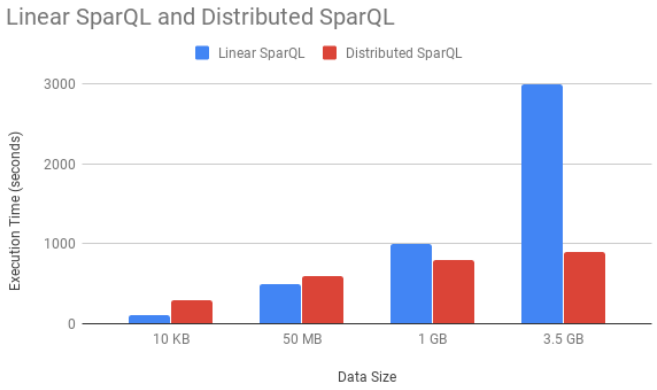

Fig. 1. Performance comparison for Linear vs. Distributed SPARQL processing with increasing data size

(1) Graph loading, (2) Basic Graph Pattern, and (3) Result calculating. The most expensive stage in terms of performance is Graph loading stage. This is expected since the first stage involves reading data from distributed storage into the memory along with all the graph creation logic. As illustrated in Fig. 2, the first stage takes almost 65% of the total execution time. Once data is in memory, the next step is to create the graph logic including the creation of *Triple Patterns*. Finally, the last step is the actual SPARQL query processing on the dataset and getting the result of the query.

These stages also present a potential improvement over the current execution pipeline. Since most of the time is spent in IO during the first stage, the system can be kept alive to keep data in memory and avoid reloading data for every query. This would greatly cut the cost of data loading, enhancing the over all query performance.

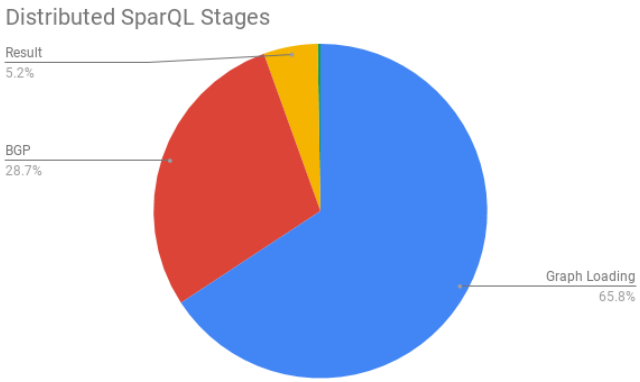

Fig. 2. Analysis of different stages for distributed SPARQL

E. Data Loading

Time spent in IO is a limiting factor for most in- memory systems. To study this, we compared the IO time for both linear and distributed implementation. Fig. 3 shows the results of this experiment. With default JVM settings, linear SPARQL failed to handle larger datasets since it resulted in out of memory exceptions. However, this did not happen in distributed setup since data was partitioned among memory in distributed nodes.

For all practical purposes, user can always tweak the JVM heap size to avoid such out of memory exceptions. However, as discussed earlier as well, although it is doable, it is not a desirable solution since it involves manual configuration on the user's end.

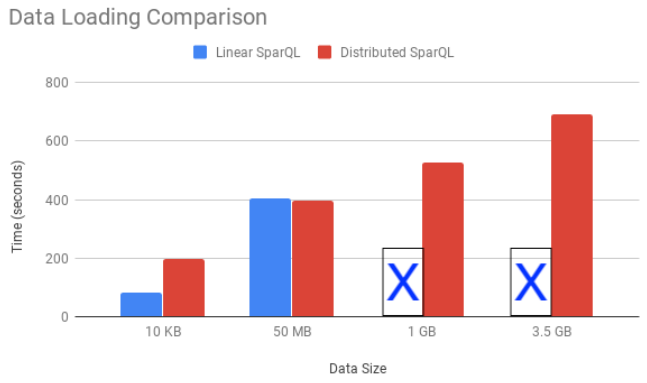

Fig. 3. Data Loading Time Comparison with default JVM settings

F. Query Complexity

Query complexity can be defined as the particular function a query performs. Although, an infinite set of queries can be designed for different datasets, we categorize them into three representative queries. The simplest query simply projected a few columns from the desired dataset. The second used an aggregation function on the dataset to compute frequency, average etc. of a particular column. The last query involved joins on different dataset columns.

In our final experiment, we compared the query performance with different complexities having fixed dataset. Fig. 4 shows the results for different sets of queries for a fixed dataset size of 1 GB. As expected, the results for simple projection and aggregation are much better for distributed SPARQL mainly because, backed distributed framework is able to work on partitioned data in parallel on distributed nodes. A linear query execution on the other hand has limited parallelization and can limited performance. However, for queries involving joins, distributed environments can limit the parallelization since records from multiple dataset partitions may potentially need to be co-grouped and joined. This involves extra IO and serialization overheads associated with moving data across network. Therefore, as shown in the Fig. 4, the performance gain for join queries is a limited as compared to other queries.

IV. Conclusion and Future Works

The obtained results showed a good query response time while using Spark based SPARQL comparing with Jena baseline performance results. We recognized that reducing the time of loading data will lead to lower the cost of potentially expensive query processing and hence improve query performance. Moreover, distributed SPARQL queries achieve better response time handling larger datasets than when running on linear SPARQL as data was partitioned among memory in distributed nodes. This is unlike the linear SPARQL which trigger out of memory exceptions. However, we can always tweak the JVM heap size to avoid such out of memory exceptions but it is still not a suitable solution as it involves manual configuration on the user's end. Since our work focus on two SPARQL queries: linear and distributed, it would be interesting in future to extend the study to investigate different types of queries as well. Furthermore, integrating parallel architecture with new settings and hardwares using different parallel RDF queries can be further investigated to bring new outstanding results in the field of semantic web.

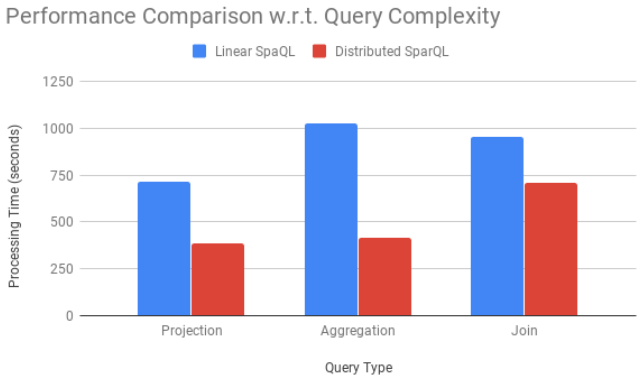

Fig. 4. Query performance comparison for different queries.

References

[1] A. Bernstein, J. Hendler, and N. Noy, "A new look at the semantic web," *Commun. ACM*, vol. 59, no. 9, pp. 35–37, Aug. 2016. [Online]. Available: http://doi.acm.org/10.1145/2890489

[2] F. Manola, E. Miller, B. McBride *et al.*, "Rdf primer," *W3C recommendation*, vol. 10, no. 1-107, p. 6, 2004.

[3] D. Brickley, R. V. Guha, and B. McBride, "Rdf schema 1.1," *W3C recommendation*, vol. 25, pp. 2004–2014, 2014.

[4] S. Albahli and A. Melton, "Triplefca: Fca-based approach to enhance semantic web data management," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 625–630.

[5] T. Chawla, G. Singh, and E. S. Pilli, "Hypso: Hybrid partitioning for big rdf storage and query processing," in *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*. ACM, 2019, pp. 188–194.

[6] H. Naacke, B. Amann, and O. Curé, "Sparql graph pattern processing with apache spark," in *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. ACM, 2017, p. 1.

[7] J. Weaver and G. T. Williams, "Scalable rdf query processing on clusters and supercomputers," in *The 5th international workshop on scalable semantic web knowledge base systems (ssws2009)*, vol. 8, 2009.

[8] H. Naacke, O. Curé, and B. Amann, "Sparql query processing with apache spark," *arXiv preprint arXiv:1604.08903*, 2016.

[9] T. Chawla, G. Singh, and E. S. Pilli, "A shortest path approach to sparql chain query optimisation," in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2017, pp. 1778–1778.

[10] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, "S2rdf: Rdf querying with sparql on spark," *Proceedings of the VLDB Endowment*, vol. 9, no. 10, pp. 804–815, 2016.

[11] G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis, and D. Plexousakis, "Rdf query answering using apache spark: Review and assessment," in *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2018, pp. 54–59.

[12] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," in *The semantic web*. Springer, 2007, pp. 722–735.