10-23-2013

D-SPARQ: Distributed, Scalable and Efficient RDF Query Engine

Raghava Mutharaju

Sherif Sakr

Alessandra Sala

Pascal Hitzler
pascal.hitzler@wright.edu

D-SPARQ: Distributed, Scalable and Efficient RDF Query Engine

Raghava Mutharaju[1], Sherif Sakr[2], Alessandra Sala[3], and Pascal Hitzler[1]

[1] Kno.e.sis Center, Wright State University, Dayton, OH, USA.
{mutharaju.2, pascal.hitzler}@wright.edu
[2] College of Computer Science and Information Technology,
University of Dammam, Saudi Arabia.
University of New South Wales, High Street, Kensington, NSW, Australia 2052.
ssakr@cse.unsw.edu.au
[3] Alcatel-Lucent Bell Labs, Blanchardstown Industrial Park, Dublin, Ireland.
alessandra.sala@alcatel-lucent.com

Abstract. We present D-SPARQ, a distributed RDF query engine that combines the MapReduce processing framework with a NoSQL distributed data store, MongoDB. The performance of processing SPARQL queries mainly depends on the efficiency of handling the join operations between the RDF triple patterns. Our system features two unique characteristics that enable efficiently tackling this challenge: 1) Identifying specific patterns of the input queries that enable improving the performance by running different parts of the query in a parallel mode. 2) Using the triple selectivity information for reordering the individual triples of the input query within the identified query patterns. The preliminary results demonstrate the scalability and efficiency of our distributed RDF query engine.

1 Introduction

With the recent surge in the amount of RDF data, there is an increasing need for scalable RDF query engines. In SPARQL, even a simple query may translate to multiple triple patterns which have to be joined. In practice, centralized RDF engines lack scalability and query performance is abridged as they are highly dependent on main memory constraints in order to efficiently process these join operations. MapReduce-based processing platforms are becoming the *de facto* standard for distributed processing of large scale datasets.

We present D-SPARQ, a distributed and scalable RDF query engine that combines the MapReduce processing framework with a NoSQL distributed data store, MongoDB. In particular, we make the following contributions:

- We describe how RDF data can be partitioned, stored and indexed in a horizontally scalable NoSQL store, MongoDB.
- We describe a number of distributed query optimization techniques which consider the patterns of the input query together with selectivity information to minimize the processing time by efficiently parallelizing the query execution.
- A comparative performance evaluation of our approach with the state-of-the-art in distributed RDF query processing.

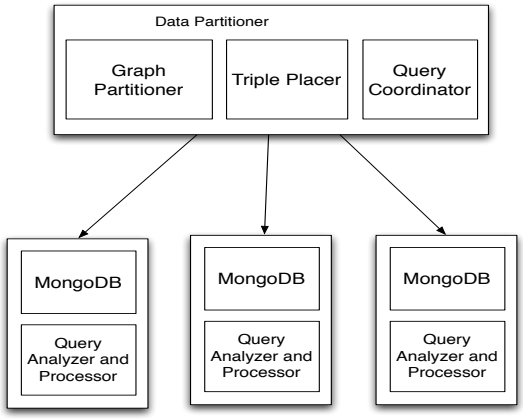

Fig. 1. System architecture

2 Approach

Figure 1 illustrates an overview of D-SPARQ's architecture. The system receives RDF triple datasets that are imported into MongoDB using a single MapReduce job which also captures all the required statistical information needed by our join reordering module in the query optimization process. A graph is constructed from the given RDF triples and using a graph partitioner [2], triples are spread across the machines in the cluster where the number of partitions is equal to the number of machines in the cluster. 'rdf:type' triples are removed from the data before partitioning the graph in order to make the graph more connected and reduce the quality of partitions. After partitioning, all the triples whose subject matches a vertex, are placed in the same partition as the vertex. A partial data replication is then applied where some of the triples are replicated across different partitions to enable the parallelization of query execution. In particular, in each partition, for all the vertices that are already assigned to that partition, vertices along a path of length n (in either direction) are added to that partition [1].

Triples assigned to each partition (machine) are stored in MongoDB[4], a NoSQL document database. In general, each RDF triple has three parts, *subject*, *predicate* and *object*. In a general key-value store, even if two parts of the triple are compressed into one, it would be less efficient to index them. Therefore, we used a document store, MongoDB, that has a good read and write speed along with good indexing, querying and sharding mechanisms. For example, in MongoDB, all triples with the same subject can be grouped in to one document. While many SPARQL queries in their basic graph patterns have a "star" pattern [3], these the patterns can share (joined on) either the same subject or object. By grouping the triples with the same subject, we would be able to retrieve

[4] `http://www.mongodb.org`

triples which satisfy subject-based star patterns in one read call. In addition, MongoDB supports indexing on any attribute of a document. It also supports single and compound indexes. We create compound indexes involving both of *subject-predicate* and *predicate-object* pairs. In MongoDB, a compound index handles queries on any prefixes of the index. For example, queries on predicate alone can be handled by the compound index *predicate-object*.

We tackle query processing by identifying the following patterns in the input query:

1. Triple patterns which are independent of each other and can be run in parallel.
2. Star patterns, i.e., triple patterns which need to be joined on the same subject or object.
3. Pipeline patterns, i.e., dependency among the triple patterns such as object of one triple pattern is same as the subject of another triple pattern (object-subject, subject-object, object-object joins).

Identifying these patterns enable us to run different parts of the query in parallel. In order to identify these patterns, an undirected labelled graph is constructed. In this graph, we find articulation points and biconnected components. In particular, articulation points provide the triples involved in pipeline pattern. A star pattern is treated as a block or a component here i.e., a star pattern cannot be split further. In general, a star pattern would have at least one articulation point and would be split up into smaller pieces if a regular biconnected component algorithm is run on it. With this tweak (keeping the star pattern as an indivisible block), all the independent star patterns can be obtained from the query graph.

In a star pattern, selectivity of each triple pattern plays an important role in reducing the query runtime. Therefore, for each predicate, we keep a count of the number of triples involving that particular predicate. For a star pattern, this information is used to reorder the individual triple patterns within a star pattern. After identifying the patterns from the query graph, processing of queries becomes a straightforward task of using querying capabilities provided by MongoDB, which automatically makes use of the appropriate indices while retrieving the records from the database. If pipeline patterns are involved in the query, care is taken to share the output of the dependent variable among all the triple patterns involved in the pipeline.

3 Evaluation

For our experimental evaluation, we used RDF datasets which are generated using SP^{2}Bench benchmark [5]. The benchmark generates DBLP data in the form of RDF triples. Our cluster consists of 3 nodes where each node has a quad-core AMD Opteron Processor with 16GB RAM and 2300MHz processor speed. MongoDB version 2.2.0 is used as a backend for our query engine. We compared our approach with the approach of [1], a distributed RDF query engine that uses RDF-3X [4] query processor as its backend. RDF-3X[5] Version 0.3.7

[5] `http://www.mpi-inf.mpg.de/~neumann/rdf3x`

#Triples	Query2		Query3		Query4	
	RDF-3X	D-SPARQ	RDF-3X	D-SPARQ	RDF-3X	D-SPARQ
77 million	217s	192.5s	80s	69.43s	OutOfMemory	319.87s
163 million	1537s	398s	434s	166s	OutOfMemory	671s

Table 1. Query runtimes (in seconds) for RDF-3X and D-SPARQ

has been used in our experiments. In particular, RDF-3X have been running on each node of the cluster and the same number of triples which are handled by our implementation are also loaded into RDF-3X on each node.

We picked three queries of the benchmark for our experiments (*Query2*, *Query3* and *Query4*). In particular, we have not considered the queries which uses the *OPTIONAL*, *FILTER*, *ORDER* features of the SPARQL query language as they are out of the scope of this paper where we are mainly focusing on the efficient execution of the join operations between the RDF triple patterns. The numbers of triples of our experimental datasets which are illustrated in Table 1 are the average number of triples loaded into RDF-3X, MongoDB of each node. So the total number of triples across all three nodes in the first case (with average of 77 million) is around 230 million and for the second case (163 million) is around 490 million triples. Each query has been executed five times and average of the runtime across all these runs has been collected. The results of Table 1 show that the query runtimes of our implementation are significantly better than that of RDF-3X, especially for larger number of triples. We observed that the performance of RDF-3X decreases with increase in the number of triples. This is a clear advantage for our query optimization techniques and the scalability of our data storage backend that relies on a NoSQL store, MongoDB.

4 Conclusion

We presented a distributed RDF query engine that combines a scalable data processing framework, MapReduce, with a NoSQL distributed data store, MongoDB. A comparative performance evaluation show that our approach can outperform the state-of-the-art in distributed RDF query processing. We are planning to continue evaluating our approach using different and bigger datasets and extend our approach to support other features of the SPARQL query language.

References

1. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB 4(11), 1123–1134 (2011)
2. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing 20(1), 359–392 (Dec 1998)
3. Kim, H., Ravindra, P., Anyanwu, K.: From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra. PVLDB 4(12), 1426–1429 (2011)
4. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. 19(1), 91–113 (2010)
5. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^{2}Bench: A SPARQL Performance Benchmark. In: ICDE. pp. 222–233 (2009)