Efficient and scalable processing of frequent SPARQL queries

Rajasekhar Velamuri
Dept. of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai, India
rajasekhar.velamuri@gmail.com

P Sreenivasa Kumar
Dept. of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai, India
psk@iitm.ac.in

ABSTRACT

A large amount of data is being published today in the RDF [6] framework using semantic mark-up. The number of triples currently published on the web is approximately 62 billion from 870 datasets [4]. We need frameworks that can query RDF data efficiently when the data sizes scale up. Incremental addition of data and computing resources is a fundamental aspect of cloud computing. We propose a framework which can run on the scale of billions of triples and answers the frequent SPARQL [7] queries in a reasonable amount of time. We integrate RDF-3X [12] (an exhaustively indexed triple store) with Hadoop [1] (a distributed data processing framework). RDF-3X has a powerful engine with a capability to pull the data fast and Hadoop gives us features such as scalability, high availability, fault tolerance and parallelism. We analyze the existing frameworks and then propose a new framework for executing the frequent queries which gives us better results.

1. INTRODUCTION

The vision of Sir Tim Berners Lee, of a world where machines could interpret the data as we humans do, gave rise to the Semantic web. In order to achieve this, Resource Description Framework (RDF) came into existence. Data in RDF is written in the (Subject Predicate Object) form popularly known as the triple format. In order to query the triple store a query language called SPARQL was standardized. A SPARQL query is made up of one or more triple patterns. A triple pattern might contain a variable or literal in the subject, predicate or object position. A query in SPARQL matches a subgraph(s) in the whole data graph. An example of this approach is given in Figure 1.

Amongst all triple stores, RDF-3X is the current state-of-the-art store. It indexes every combination of subject, predicate and object using clustered B^+-tree indexes. It implements query optimization techniques for fast query execution. Thus RDF-3X is 1-2 orders of magnitude faster than existing triple stores.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
The 19th International Conference on Management of Data (COMAD), 19th-21st Dec, 2013 at Ahmedabad, India.
Copyright ©2013 Computer Society of India (CSI).

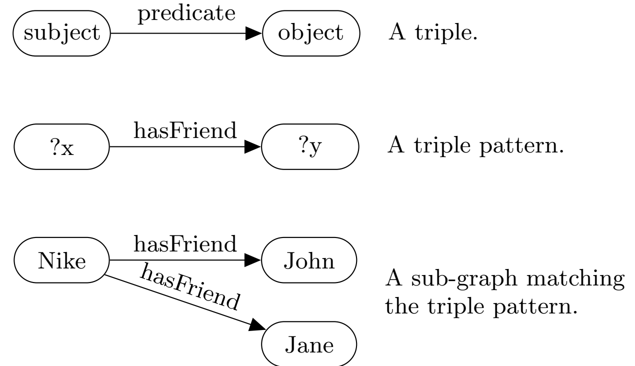

Figure 1: SPARQL Example

Massive amount of data on the semantic web gives rise to distributed repositories. The distributed, parallel data processing framework Hadoop has the capability of scalability, high availability and fault tolerance. It uses the map-reduce style of data processing. A possible integration of RDF-3X and Hadoop can give us best of both the worlds, namely efficiency and parallelism.

In a query log there are always some frequent triple patterns. Frequent triple pattern(s) can be pre-computed so that when the frequent triple patterns are posed again we can answer the query fast. We can cache the results of these frequent triple patterns. Thus we can avoid joins which are done as part of the hadoop job in case of any query. The joins in this case are still done by RDF-3X database locally on each node. Data gets accumulated over time and the frequent triple pattern(s) can change. Thus we also provide an update algorithm which can update our frequent databases when either more data gets added or frequent triple patterns change. In our approach we perform better than Tao Yang et al.[8] as we avoid the unnecessary shuffle and reduce phase of a hadoop job and also avoid considering the triples which do not contribute to the frequent triple pattern(s) result.

The contributions of our work are summarized below:

- A triple reconstruction based partitioning algorithm.
- A seperate partition for frequent queries data.
- A data update algorithm.

In Section 2 we discuss the work related to querying and frequent triple pattern(s) querying. In Section 3 we discuss

architecture of the proposed system. In Section 4 we discuss the proposed data partitioning approach. In Section 5 we give the experimental results of the proposed approach as compared to Tao Yang et al. In Section 6 we give the conclusion and future work.

2. RELATED WORK

In this section we discuss the existing methods for data partitioning and querying. Bhavani Thuraisingham et al. [11] propose a heuristic based approach for SPARQL querying in the cloud. Since there are no indexes in HDFS [2] they suggest a *predicate* and *predicate_object* split based partitioning technique. Another approach is the one suggested by Abadi et al. [10] and later by Tao Yang et al. which is the one we adapted for our work. The framework we use is suitable for querying giant graphs due to the the fault-tolerance and scalability. As stated earlier querying can be made efficient when complemented by an efficient data partitioning mechanism.

Abadi et al. propose data partitioning approaches called hash partitioning and graph partitioning. Hash partitioning, has the disadvantage that it partitions according to the vertices and not according to the graph topology. Graph partitioning takes care of the above loophole and does the partitioning according to the graph topology such that the neighbouring nodes are present on the same partition. The 2-*hop* approach proposed by Abadi et al. works well for the LUBM [13] benchmark as the maximum number of joins required by any query in the LUBM benchmark are covered within 2-hops. This approach fails if there are more number of joins.

Tao Yang et al. suggest an approach where they select the frequent triple patterns from a query log and distribute the sub-graphs of these frequent triple patterns across the hadoop cluster where they are stored on the local RDF-3X database. The user query is run on all the nodes. We follow the approach taken by Tao Yang et al. We present in Table 1 the approximate amount of data contributed by the result sets of the frequent triple patterns as compared to the whole data for 0.001, 0.01, 0.1 and 1 billion triples.

Table 1: Data contributed by the frequent triple patterns.

No. of Triples	Total data	Frequent triple patterns data
1 billion	225GB	500MB
0.1 billion	25GB	50MB
0.01 billion	2.5GB	5.5MB
0.001 billion	250MB	550KB

3. ARCHITECTURE

We define some terms before describing our architecture.

Frequent query (FQ): A query in the query log consisting of frequent triple pattern(s).

Infrequent query (IFQ): A query which is not frequent is an infrequent query.

Frequent Query Answer Tuple (FQAT): A tuple which appears in the result set of the frequent queries.

Frequent Query Database (FQDB): A database which consists of triples contributing to the frequent query answer.

Infrequent Query Database (IFQDB): A database which answers infrequent queries.

Frequent Query Result Set (FQRS): A result set obtained by executing any of the frequent queries is a FQRS.

The architecture consists of a data partitioner, a data loader and a query processor. The (in)frequent triple pattern(s) data is present in the (I)FQDB. The input dataset is loaded into a single instance of RDF-3X. Each frequent query is executed on this database to generate FQRS. The data loader gathers all the FQRS and runs the data placement algorithm suggested by Tao Yang et al. The datasets generated are loaded into FQDB. The same input dataset is partitioned into a number equal to the number of nodes in the cluster and loaded into the IFQDB. The query with non-frequent triple patterns is executed on these IFQDB using the algorithm proposed by Bhavani et al.

The query processor checks whether a query is a frequent query (FQ) or not. In the case of frequent queries, we bypass the hadoop framework and the query is executed on the nodes locally using JSCH [3] middleware. We aggregate the data after all the queries have been executed. The JSCH middleware can be made fault tolerant by giving the list of the nodes where the replica of the data is present. Thus we can also ensure the availability of data.

Below is a pictorial representation of our system architecture.

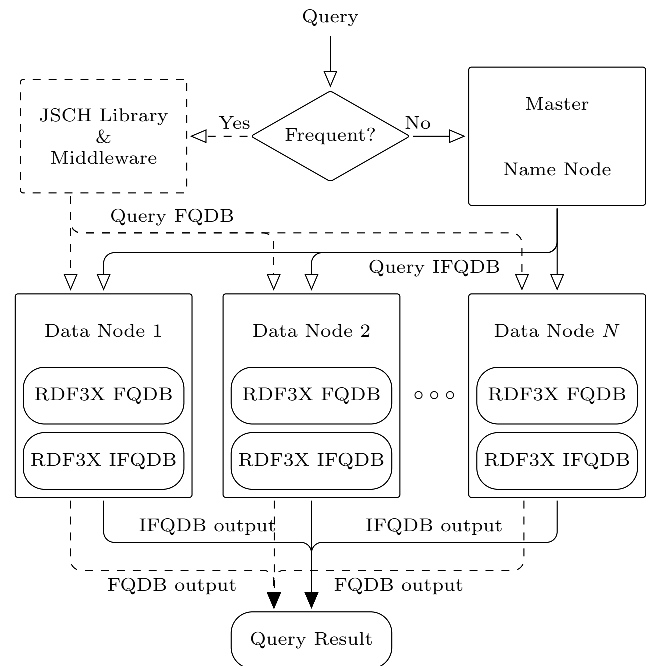

Figure 2: Architecture diagram

4. DATA PARTITIONING

Before the start of the data partitioning mechanism we would like to introduce a few terms for the convenience of the discussion.

query: A FQ which is also a SPARQL query.

queryList: List of all the FQ.

tuple: A subgraph contributing to the Result Set.

resultSet: The result of a FQ executed on a FQDB

triplePattern: A triple pattern in a FQ.

We first load the whole dataset on a single instance of RDF-3X. For every query in queryList we execute the query on the RDF-3X database and generate the resultSet. Each tuple in the resultSet is used to construct the triples which have contributed to the formation of this tuple. All these triples belong to the first node in the cluster. Triples generated from the next tuple belong to the second node. The above process is repeated till all the tuples in the resultSet are distributed in a round-robin fashion. Round-robin partitioning gives balanced partitions on which we run the data placement algorithm suggested by Tao Yang et al. All the partitions for a particular node are aggregated and loaded into the FQDB for that node. We present an example below for the convinience of the discussion. Consider a frequent query (FQ) generating resultSet which consists of 2 tuples.

```
SELECT ?X ?Y ?Z
WHERE { ?X hasBrother ?Y. ?X hasFather ?Z }

<John> <Nick> <Mike> , <Ron> <Harry> <Adam>
```

The triples generated by the Data-Partition algorithm for the first(second) tuple belong to the first(second) nodes.

```
<John> <hasBrother> <Nick> , <John> <hasFather> <Mike>
<Ron> <hasBrother> <Harry> , <Ron> <hasFather> <Adam>
```

Thus, for every tuple we do the distribution of triples in a round-robin fashion. This process is repeated for all the queries. Thus for the frequent queries we search only for the limited triples which contribute to the frequent query leading to small query execution times.

```
Data-Partition(dataSet, queryList, partitionCount)
1  load dataSet into RDF-3X database
2  for query in queryList
3      resultSet = run query on RDF-3X
4      for tuple in resultSet
5          partition = round-robin(partitionCount)
6          for triplePattern in query
7              build triple from tuple, triplePattern
8              place triple into partition.next()
```

The Data-Partition algorithm above is for a single node. According to Bhavani Thuraisingham et al., when the number of triples beyond 1 billion were loaded into the RDF-3X database, it failed to answer few queries with many joins from the LUBM benchmark. Hence we can use the map-reduce [9] based distributed data partitioning algorithm for large datasets. In the map-reduce version of the algorithm, the input dataset is split and randomly loaded into all the nodes. It uses the iterative map-reduce based algorithm proposed by J Myung et al.[14].

5. DATA UPDATION

We would like to introduce a few terms which are used in the Data-Update algorithm.

NQueries: Frequent triple patterns in the new query log.

OQueries: Frequent triple patterns in the old query log.

CommonQueries: Frequent triple pattern(s) common to both the new query log and the old query log.

NewInfreqQueries: Frequent triple pattern(s) from the old query log which are no more frequent.

NewFreqQueries: The frequent triple pattern(s) present in the new query log and absent in the old query log.

Below we propose the Data-Update algorithm. Data gets accumulated over time and new frequent triple patterns are found. Thus we have 3 scenarios. 1. New data and new frequent triple patterns. 2. New data and old frequent triple patterns. 3. Old data and new frequent triple patterns.

```
Data-Update(Data, NQueries, OQueries, PCount)
1  split Data into N partitions
2  load N partitions into N IFQDB
3  CommonQueries = NQueries ∩ OQueries
4  NewInfreqQueries = OQueries \ CommonQueries
5  NewFreqQueries = NQueries \ CommonQueries
6  if CommonQueries.size == 0
7      delete FrequentDB
8  for query in NewInfreqQueries
9      for TriplePat1 in query
10         convert literals in TriplePat1 into variables
11         for TriplePat2 in CommonQueries
12             if TriplePat1 ≠ TriplePat2
13                 delete triples from FQDB
14 for query in NewFreqQueries
15     resultSet = run query on IFQDB[a]
16     for tuple in resultSet
17         partition = round-robin(partitionCount)
18         for triplePattern in query
19             build triple from tuple, triplePattern
20             place triple into partition.next()
```

[a] Using iterative map-reduce algo by Myung et al.[14]

In the first case either we find a few new FQs or all FQs are replaced, in which case all FQDBs become invalid and all the triples from the FQDBs are deleted. The new dataset is partitioned into N parts where N is the number of nodes. We then randomly load these datasets into the existing IFQDB. We make a basic assumption that the new FQs are different from the existing FQs. Let us assume that there are m new FQs where $m \leq M$ and M is the set of FQs present earlier. We delete the data for the *M-m NewInfreqQueries* queries. For each of the queries which are getting replaced from the frequent queries list, we check the triple patterns present in a query and compare it with each of the triple patterns of the queries which are still on the frequent queries list. A match means the triples corresponding to that triple pattern are retained else the triples are deleted. This process essentially deletes all the data for the *NewInfreqQueries*. Since we keep the FQDB isolated from the IFQDB, updation of triples happens to the FQDB while the insertion of new data happens to the IFQDB. We now run the map-reduce job which gives us the result set for the current query. We repeat this process for all the queries to generate m datasets. We run a modified version of the Data-Partition algorithm which gives us $N*m$ datasets. We apply the LNS (Largest Neighbourhood Search) based approximation algorithm proposed by Tao Yang et al. which generates $N*m$ datasets which have minimum overlap. We generate the original triples from

this result set and add them in a round-robin fashion to the FQDBs. Thus the current FQDBs is according to the current list of FQs. In the last case the data from FQDBs is deleted and the above steps are repeated without any addition to IFQDB.

6. EXPERIMENTAL ANALYSIS

Node configuration: We use a cluster of 4 nodes. Each node runs an Intel i5 processor with a 512 GB hard disk, 4 GB RAM and 3.0 Ghz clock speed. The data partitioning is done on a node running Intel Xeon processor (E5-2650) with 2 TB hard disk, 32 GB RAM and 2.00 Ghz clock speed.The OS used on all the nodes is Ubuntu version 12.04 LTS.

Software installed on every node: We use the version 3.7.0 of the RDF-3X database, the version 20.205 of Hadoop and the version 1.7 of java in order to run our experiments.

Dataset used for comparison: We use the LUBM benchmark to test our approach against the one suggested by Tao Yang et al. As suggested by Bhavani Thuraisingham et al., the queries 1, 2, 4, 9, 12 and 13 of the LUBM benchmark are sufficient to test the performance of any framework. Each query is run 10 times and the query execution times are averaged.

We report 3 results namely FQs run according to the approach of Tao Yang et al., FQs run on only FQDB using map-reduce (Proposed approach MR) which differs from Tao Yang et al. in that we run map reduce on FQDB only. and FQs run on FQDB using our architecture (Proposed approach). In the table reported below the query Q2 from the LUBM benchmark takes more time in the case of Tao Yang et al. which is due to the fact that during execution 4 map tasks failed and were re-issued by Hadoop framework for completing the query successfully. In the case of LUBM benchmark the query Q9 is taken as of utmost importance because of the number of joins involved. We can see from Table 2 that we actually get a 76X speedup on this query. On the rest of the queries we get more than 32X speedup. We do not compare against query Q2 as we think it to be unfair to judge our framework on it because of the above mentioned reason.

The comparatative analysis of the results of different approaches for 10000 universities (1 billion triples) are presented below in Table1 respectively.

Table 2: Query exec. time (in Sec.) for 1 billion triples.

Approach	Q1	Q2	Q4	Q9	Q12	Q13
Proposed approach	<1	<1	<1	<8	<1	<1
Proposed approach MR	32	32	32	60	31	31
Tao Yang et al.	32	2192	32	611	32	32

7. CONCLUSIONS

We present a framework for querying a billion triples using a frequent triples patterns based approach. In this approach for frequent triple pattern(s) joins are done by RDF-3X, hence the execution times are fast. We are also able to avoid the shuffle and the reduce operations resulting from a hadoop map-reduce job in the case of a frequent triple pattern(s) which makes our execution time better than the current state-of-the-art framework. In the current version we are addressing only the frequent triple pattern(s) querying. In future we would like to address the infrequent triple pattern(s) querying. The best framework for the infrequent triple pattern(s) till date is the one presented by Abadi et al. This approach introduces a lot of redundant data when the average degree of the vertex becomes high. For a given query Q with N joins, we have to take a *N-hop* approach to satisfy the query. More hops only increase the redundant data present on a node.

8. REFERENCES

[1] HADOOP http://hadoop.apache.org/.

[2] HDFS http://en.wikipedia.org/wiki/apache_hadoop.

[3] JSCH http://www.jcraft.com/jsch/.

[4] LOD http://stats.lod2.eu/.

[5] OWL http://www.w3.org/tr/owl-ref/.

[6] RDF http://www.w3.org/rdf/.

[7] SPARQL http://www.w3.org/tr/rdf-sparql-query/.

[8] Tao Yang J. Chen X. Wang Y. Chen and X. Du. Efficient sparql query evaluation via automatic data partitioning. *DSAA*, vol. 7826:244–258, 2013.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Proc. Sixth Conf. Symp. Operating Systems Design and Implementation*, 2004.

[10] J. Huang and D. J. K. Ren. Scalable sparql querying of large rdf graphs. *Proc. VLDB Endowment*, vol. 4 no. 11:1123–1134, 2011.

[11] M. Husain J. McGlothlin M. M. Masud L. Khan and B. Thuraisingham. Heuristics based query processing for large rdf graphs using cloud computing. *TKDE*, vol. 23 no. 9:1312–1327, September 2011.

[12] T. Neumann and G. Weikum. Rdf-3x: A risc-style engine for rdf. *Proc. VLDB Endowment*, vol. 1 no. 1:647–659, 2008.

[13] Y. Guo Z. Pan and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3:158–182, 2005.

[14] J. Myung J. Yeon and S.g. Lee. Sparql basic graph pattern processing with iterative mapreduce. *In Proc. of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC:6:1–6:6, 2010.