**CS5574**

**Phase II: Project Report**

*Anas Katib Imrul Siddiuqe Suresh Yarra Prudhvi Atluri*

**1. Introduction**

This report documents the newly introduced design and implementation concepts of  our approach to utilize a NoSQL document store (i.e. monogDB) to index RDF data and process SPARQL queries (please refer to the Research Proposal for more details about the system’s design). In addition, the report presents brief evaluation of our implementation and the task distribution among team members.

**2. Design**

Our main system’s design is consistent with our initial proposal. It is composed of a database builder and a query processor. The database builder is responsible for converting the N-triple data into documents, and storing these documents in a mongoDB database. On the other hand, the query processor is responsible translating SPARQL queries into mongoDB queries and executing the resulting queries.

Following the steps of Mihaela et al. [1], we have originally planned to group RDF graphs based on the shared predicate to speed up the query processing time when the predicates are known. Utilizing such approach is possible to in mongoDB, however since mongoDB returns the whole document containing a match for a given query, our focus would be diverted from parsing and processing SPARQL queries to parsing and manipulating mongoDB documents.

In consequences, to avoid any deviation from our main task, we have modified our approach to store RDF graphs such that each RDF triple is stored as an independent document in mongoDB. Thus, monogoDB will only return the subset of documents that matched a given query (not the whole document whose content have matches and non-matches).

Since RDF is represented as a graph, and SPARQL queries can be thought of as graphs too (i.e. sub-graph that is present in actual data), we have relied on generating the query graph to process the query triples. However, we introduce minor modifications on the query graph in order to able to process the query appropriately. The modified resulting graph is called a *data guide* (DG) *graph*.

The purpose of the data guide graph, therefore, is to find an optimal order such that, triples that are independent should be processed beforehand. This will also guarantee least amount of intermediate data are processed. Eventually the query executes faster. Essentially, the DG graph is considered the query plan in our proposed system. The following sections describe how a DG gets created and how we traverse it.

Let’s consider the two following queries:

|  |  |
| --- | --- |
| Q1. | Q2. |
| SELECT ?x, ?y, ?z  WHERE ?x <name> “Jack”.  ?x <friend> ?y  ?y <brother> ?z  ?x <home> “Kansas City” | SELECT ?X ?Y ?Z  WHERE ?X ?Y ?Z  ?X <name> “Jack”  ?Z <name> “Adam” |

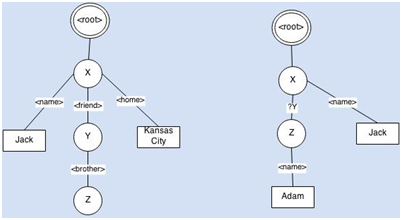


Figure 1: DG for Q1 (left) & Q2(right)

As we can see the above DGs will be formed for Q1 and Q2 which are essentially trees. We added a root node that indicates the starting point of the tree. The steps to create a DG are as follow:

1. Create a root node.
2. Read the first condition and create a edge from root to subject and another from subject to object. The value of the edge is the predicate. Also keep track of the variables that have been visited.
3. While processing a condition, if we encounter a variable in the object position which has been used previously as an subject, then we transfer that sub-tree under current subject. For example, in Q1, let’s say, Y <brother> Z is processed before X <friend> Y and that there was a edge from root to Y. Now when we find X <friend> Y, we create an edge from X to Y and transfer the Y-edge-Z sub-tree under X-edge-Y.

When the DG is created for given query, we can start processing/translating each triples according to the DG:

1. Compare and sort the triples. To compare the two triples the following comparison rules will be applied:
   1. Ancestor in DG should be processed first
   2. Condition with less variables should be processed first
2. Execute the sorted query triples.
3. For each triple (e.g. Y <brother> Z) we assume that we already have a set of values for Y and replace Y with these values. If Z isn’t found for any Y, then we track back until we reach root and then discard that combination path from root. This operation will guarantee that all the paths from <root> to Z are valid eventually. Suppose in Q1 we found an X who has a friend Y but Y don’t have a brother Z. So we shouldn’t consider that (X,Y) pair at all, hence we remove that combination.
4. When all conditions are executed we just traverse from root to leaf for each (x,y,z) combination.

**3. Implementation**

**3.1 Database Builder**

As discussed in Design section, we considered a few possible structures to store the data in MongoDB data store. Since for each match the mongo query return the pointer on the whole document instead of specific record, we preferred to store each RDF tuple in one mongo document as shown in the Figure 2. This section describes how we stored the RDF N-triple data in the MongoDB format (more details in Appendix A).

<sub1> <prop1> <obj1>

<sub1> <prop2> “val”

<sub2> <prop1> <obj1>

{ subject: <sub1>

property: <prop1>

object: <obj1>

}

{ subject: <sub1>

property: <prop2>

object: “val”

}

{ subject: <sub2>

property: <prop1>

object: <obj1>

}

Figure 2: RDF triple to Mongo Document

The database builder is a program that takes RDF as input and generate corresponding Mongo document as output and stores it into the configured MongoDB data store. The following code snippet illustrates how we can connect to local mongo store and set the appropriate settings:

MongoClient mc=new MongoClient(Arrays.asList(

new ServerAddress("localhost", 27017)));

DB db=mc.getDB("suresh");

Set<String> sc=db.getCollectionNames();

DBCollection c=db.getCollection("col");

The above code will use the database ‘suresh’ and it will get all the collections in that database and it uses the ‘col’ collection. Note that, even if the database “suresh” isn’t allocated before running the program, mongo will create one while the program runs.

The next code snippet shows how we create a Basic object (that essentially is an document) from the RDF triple and store it in the collection (please refer to the source codes for more details). We repeat the process for all the RDF triples.

BasicDBObject bob=new BasicDBObject("subject", q[0])

.append( "property", q[1]).append("object", s);

c.insert(bob);

**3.2 Query Processor**

Although the goal of the query processor is to translate SPARQL queries into MongoDB queries and execute the resulting translation, there are several steps that take place before that stage can be reached. In this section we will cover the implementation details of these steps: parsing the SPARQL query, generating the query plan, translating and executing the query plan.

We have resorted to Apache Jena ARQ [2] to parse the SPARQL input. Specifically, the ARQ parser retrieves output variables and the query triple patterns and forwards them to the query planner. The task of the query planner is to generate a data guide graph that will represent a plan to translate the query patterns and execute the MongoDB queries. The generated data guide will be stored in a hybrid graph data structure that is made of an incidence list to store edge objects with each incident vertex having an adjacency list of its connected vertices [3].

Once the data guide graph is generated, the query processor will traverse it to process the corresponding queries. As an edge is being traversed, its corresponding vertex-to-vertex relationship will be translated into a MongoDB query and the intermediate results will be stored into a dynamic matrix of values as rows and query variables as columns (i.e. dMat). Initially, dMat is empty but once a query pattern is translated and executed, the results will be inserted into the matrix. Then for every row in the matrix the query processor will translate and execute the next relationship, which will have three outcomes:

* 1. *no match*; then delete the row form the dMat.
  2. exactly *one match*; then if the relationship contains a new variable insert a new column for that variable and insert its value in the current row, but do not create a column if there was no new variable.
  3. *1 < x matches*; then replicate the row x-1 times and if the relationship contains a new variable insert a new column for the variable and insert its value in the current row and in the replicated rows, but do not create a column if there was no new variable.

Once all edges or relationships have been traversed the querying phase is over and the columns of dMat that correspond to output variables can be outputted.

**4. Evaluation**

In order to confirm the correctness of our implementation we have utilized Apache Jena [2] to run the supplied query test cases, and we had the output of each query saved to a file. Then we ran the queries using our approach and we visually compared the results to confirm their validity.

In the early stages of evaluation, the system was not successful in passing several test cases; although its output contained correct results, there were additional duplicates. After several trials, we have discovered that the existence of duplicate MongoDB documents in the database was causing our output to have duplicate results. Therefore, we removed the duplicate documents from the database and consequently the system passed most test cases after debugging.

**5. Test Case**

However, we faced a test case (i.e. query 7) where our system would run out of heap memory before it was to complete the processing task. Nevertheless, we were able to overcome that issue by allocating more memory resources to the program (specifically increasing the JVM heap memory to 1GB). After that, the system passed all test cases successfully.

**6. Task Distribution**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task | Anas | Imrul | Suresh | Prudhvi |
| Database Builder |  |  | \* | \* |
| Query Processor | \* | \* |  |  |
| Jena Evaluator | \* | \* | \* | \* |
| Documentation | \* | \* | \* | \* |

**7. References**

1. Bornea, Mihaela A., et al. "Building an Efficient RDF Store Over a Relational Database".2013. 121-132. Print.
2. Apache Jena. The Apache Software Foundation, Web. 01 May 2014. <https://jena.apache.org/index.html>.
3. Weiss, Mark A. Florida International University. School of Computing and Information Sciences, Web. <http://users.cis.fiu.edu/~weiss/dsj2/code/Graph.java>.

**8. Appendix A:**

1. We used following online resources to install MongoDB: [www.mongodb.org/downloads](http://www.mongodb.org/downloads), <http://docs.mongodb.org/manual/installation/>. MongoDB is stored locally and port 27017 was used.