CS 5574
Large Scale Data Management
Phase I: Project Proposal

An Approach for RDF indexing and
SPARQL Query Processing Using NoSQL

Anas Katib Imrul Mukit Suresh Yarra Prudhvi Atluri

Introduction

The ever-growing volume of data in recent years is posing several challenges and concerns about handling big amounts of data (in the order of terabytes). As we observe the nature of such data, it is clear that there is a strong need for finding an alternative to the traditional relational database management system (RDBMS). Big data is heterogeneous and difficult to efficiently bind to a fixed schema. Its growth rate is increasing constantly, as it could be composed of multiple levels of nested elements that cover a tremendous range and depth. Such a nature adds a huge burden on the server-to-server communications and increases the complexity of processing various queries (negatively impacting the performance).

Concerned with the issues that arise when dealing with such unstructured data, researchers came up with a new database architecture called: NoSQL (i.e. Not only SQL). NoSQL is not only capable of handling the nature of the data we have introduced, but also capable of handling data that is in the form of tables and spreadsheets. Nevertheless, some researchers argue that the maturity of RDBMS makes it a preferable choice when dealing with structured data. In general NoSQL databases are categorized into four categories: key-value, column family, document, and graph stores. One's selection of any category is dominated by the requirements and the nature of the data he or she is dealing with.

On the other hand, one of the forms that have been developed to represent data is the Resource Description Framework (RDF). RDF excels in capturing information and representing data. However, it is still difficult to query RDF graphs efficiently (when queries involve joining multiple variables). A wide variety of research has been conducted to determine the optimal method to index and query RDF data. In this project, we propose our unique approach to accomplish this task aiming to outperform other solutions in this area. Our approach to index RDF graphs and process SPARQL queries utilizes the document-oriented NoSQL database technology. Despite the fact that native graph stores can easily store RDF data (i.e. graphs), we will try to utilize the document-based store to reduce the complexity of query processing so that we could achieve better performance.

Design

Our approach to index and query RDF data using a document-oriented data store is influenced by the work of Mihaela et al. in their attempt to build an RDF store on top of an SQL database [1]. In particular, they emphasize the grouping of RDF graphs based on the predicates (in addition to the subjects) to utilize "the advantages of traditional relational representations" [1]. In addition, our implementation is also influenced by the optimized index structures proposed by Harth and Decker [2], who argue that the performance advantages one acquires from utilizing their proposed indexes outweigh the increased index space associated with having additional indexes.

The authors of both works have proposed sophisticated approaches to overcome the issues at their hands. However, not all of the attractive concepts in the publications are applicable to our problem. Therefore, our proposed approach deviates from the solutions proposed in these publications, yet our system's architecture is closely related. Furthermore, our proposed solution consists of two main components: a database builder and a query processor (Figure 1). The database builder is responsible for converting the RDF data (given in N-Triples format) into database documents and storing these

documents in the database server. On the other hand, the query processor is responsible for translating SPARQL queries into corresponding query-language queries and returning the query results.

In order to reduce the complexity of our solution and to enable easier integration, we designed our system aiming for higher cohesion and looser coupling. Each of the two main components is composed of an independent and essential set of modules that accomplishes the main task. The database builder consists of two modules: the N-Triple parser (NT parser), which parses N-Triples and converts them into database documents, and the database builder communication unit (DB Builder COM), which is responsible for setting up the database and storing the produced documents. We refrained from using an external software tool when parsing the data to automate the progress of the interaction and enable us to modify the parser as appropriate.

Similarly, the query processor is composed of three modules: SPARQL parser, query builder, and query processor communication unit (QP COM). The SPARQL parser is responsible for extracting important information from the input query (e.g. return variables). The query builder is responsible for generating a query plan based on the extracted information by the SPARQL parser. In addition, it is responsible for generating the appropriate query (based on the generated plan). On the other hand, the QP COM is responsible for initiating the generated query and retrieving the query results.

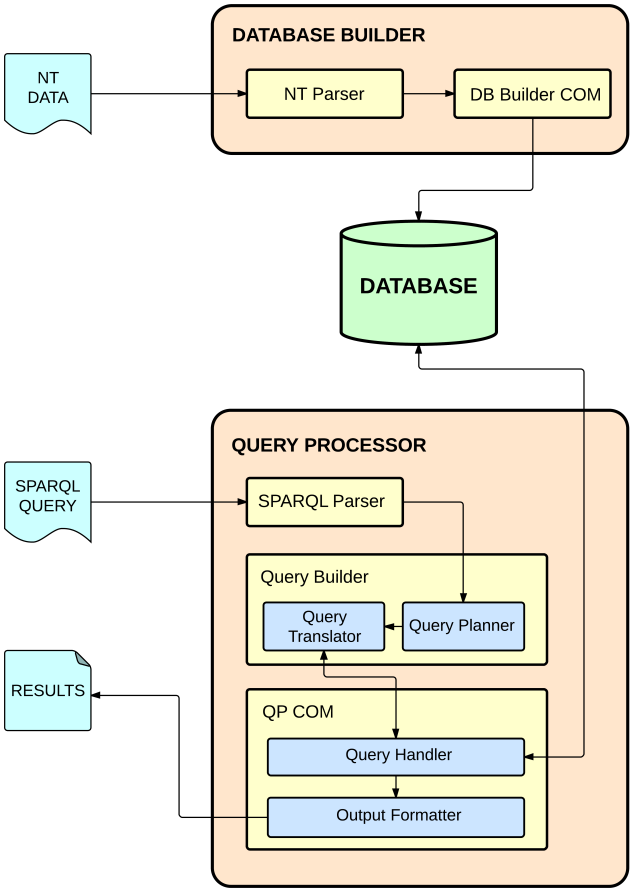

Figure 1. System's Architecture

Implementation

Database

In our implementation of the system we chose MongoDB as our NoSQL, document-oriented database facility due to several reasons. Firstly, it is one of the popular databases in terms of scalability and performance. Secondly, it has good querying and indexing capabilities, which makes it a strong and flexible tool at the same time. In addition, unlike most RDBMS, MongoDB doesn't require JOINs when processing queries. Instead, it uses the map/reduce "paradigm for condensing large volumes of data into useful aggregated results" [3]. MongoDB's documents can be represented by JSON formatted text, which is simple to construct and manipulate.

Each document that is stored in MongoDB must have a key id that uniquely identifies the document in the database. However, the documents' content does not have to conform to a specific schema. Furthermore, documents can have embedded sub-documents and references (relationships) to other documents. Additionally, documents are indexed by default based on their key ids, yet multiple indexes can still be constructed and adopted by the user.

Database Builder

From our initial investigation, we identified that predicates in RDF N-Triples will be the most suitable candidate for the document key id. Therefore, when the NT parser processes the RDF data it will construct the JSON documents and assign the predicates as document key ids (such that the index will function as a predicate-to-document lookup map). All the triples having same predicates will be grouped together in the same document. Inside the document, the (subject, object) pair of a triple will be set as a JSON attribute-value pair (Figure 2). To implement this task, the NT parser will read the RDF data, sort the triples by their predicate, and write them periodically to the appropriate document.

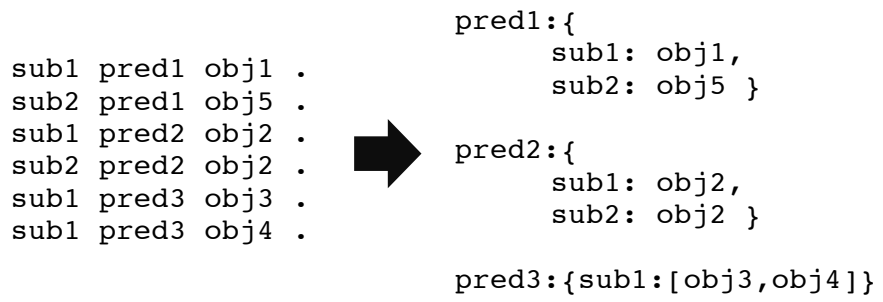

Figure 2. Forming JSON documents (right) from NT data (left).

After all the documents have been created, the DB Builder COM will connect to the database, store the documents, and create indexes. In order to leverage the relative redundancy of predicates in the RDF data and the relative uniqueness of objects in SPARQL queries, we chose to have two indexes in our implementation: one default index for predicates and one index for the objects.

Query Processor

The main task of the query processor is to parse the SPARQL query and translate it into an appropriate MongoDB query. The SPARQL parser will identify the return variables as well as the relation(s) between different query variables and literals. Furthermore, to simplify the parsing process, the parser will ignore the annotation attached to literals (we are assuming that annotations will not change the actual final output). Moreover, relaying on the information passed by the SPARQL parser, the query planner will identify the joins between two or more triples, create a query plan that specifies the optimal order of query translation, and forward the plan to the query translator. The query translator will then break the query plan down to atomic query statements and utilize a query handler to execute the

statements sequentially, where each following statement will be performed on the previous result set. After all statements have been executed, the output formatter will convert the JSON output from the query to a standard SPARQL query output. Having the output in SPARQL format will help us confirm the correctness of our solution for any evaluation we conduct. For evaluation and testing purposes, we will develop a Java program that uses the Jena library to run SPARQL queries on the RDF dataset. The results produced by Jena will be the reference point that we will use to evaluate our approach.

Task distribution

The development of the two main components (i.e. DB builder and query processor) can go in parallel. Also, the output evaluator has to be developed as early as possible. Therefore, we are considering the following task distribution among team members:

#	Task	TM1	TM2	TM3	TM4
1	Investigate Parsing techniques	*	*		
2	Database Builder				
	NT Parser			*	
	DB Builder COM			*	
3	Query Processor				
	SPARQL Parser	*			
	Query builder	*	*	*	*
	QP COM	*	*	*	*
4	Evaluator				*

Schedule

For the convenience of the development process, we will establish three milestones:

Milestone	Date	Task #
1	March 7	2.1, 4
2	March 14	1, 2.2
3	April 4	3

We are expecting that a reasonable parsing technique would be identified by the time we reach the first milestone. However, while developing the SPARQL parser, we might discover some issues with our parsing approach. Therefore, we have postponed the deadline to reach a suitable parsing approach to the second milestone. Keeping in mind that we have to finish the project within Phase II, we plan to finish the third milestone one week earlier than the due date.

References

[1] Bornea M, Dolby J, Kementsietsidis A, et al. Building an efficient RDF store over a relational database. *Proceedings of the 2013 ACM SIGMOD International Conference on management of data.* ACM; 2013:121-132.

[2] Harth A, Decker S. Optimized index structures for querying RDF from the web. Proceedings of the Third Latin American Web Congress, LA-WEB 2005 Vol 2005. ; 2005:71-80.

[3] "Map-Reduce." *MongoDB*. MongoDB, Inc., Web. 21 Feb. 2014. <http://docs.mongodb.org/manual/core/map-reduce/>.