# RDFChain: Chain Centric Storage for Scalable Join Processing of RDF Graphs using MapReduce and HBase

Pilsik Choi[1,2 *], Jooik Jung[1] and Kyong-Ho Lee[1]

[1]Department of Computer Science, Yonsei University, Seoul, Republic of Korea
pschoi@icl.yonsei.ac.kr, jijung@icl.yonsei.ac.kr,
khlee@cs.yonsei.ac.kr
[2]Mobile Communication Division, Samsung Electronics
pilsik.choi@samsung.com

***Abstract.*** As a massive linked open data is available in RDF, the scalable storage and efficient retrieval using MapReduce have been actively studied. Most of previous researches focus on reducing the number of MapReduce jobs for processing join operations in SPARQL queries. However, the cost of shuffle phase still occurs due to their reduce-side joins. In this paper, we propose RDFChain which supports the scalable storage and efficient retrieval of a large volume of RDF data using a combination of MapReduce and HBase which is NoSQL storage system. Since the proposed storage schema of RDFChain reflects all the possible join patterns of queries, it provides a reduced number of storage accesses depending on the join pattern of a query. In addition, the proposed cost-based map-side join of RDFChain reduces the number of map jobs since it processes as many joins as possible in a map job using statistics.

**Keywords:** Map-side join, chain centric storage, HBase, NoSQL, RDF, SPARQL, MapReduce, Hadoop

## 1 Introduction

As an enormous amount of Linked Data is available, processing a SPARQL query into a massive RDF dataset becomes a challenging task when scalability and performance issues are taken into consideration. Progress in many researches into SPARQL query processing has been made with the use of MapReduce, a distributed parallel processing framework. In particular, Hadoop[1] is the most popular open source version of MapReduce. Particularly, the conventional MapReduce-based join processing methods are divided into two approaches: reduce-side join and map-side join [1]. Map-side join outperforms reduce-side join since the shuffle and reduce phases of reduce-side join are not required. However, map-side join requires the datasets to be equally partitioned by join keys. It is not just non-trivial but demanding for the condition to be met in the case of multi-way joins. Przyjaciel-Zablocki et al. [2] have pro-

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

posed the Map-Side Index Nested Loop Join (MAPSIN) using HBase[2], which is a distributed, scalable and column-oriented NoSQL storage. HBase is well suited for random access due to the sparse multidimensional sorted map, and is also appropriate for a data model that requires processing row keys such as table scans and lookups. MAPSIN provides an optimized algorithm for reducing the number of storage access for star pattern joins, but not for chain pattern joins. Therefore, we propose RDFChain with the following contributions:

- RDFChain reflects every possible join patterns in its storage schema. Specifically, the proposed chain centric storage, which reflects relations among the subjects and objects of RDF triples, reduces the number of storage access.
- RDFChain reduces the number of map jobs in multi-way joins. RDFChain estimates the cost of join processing using statistics to split a query. The queries separated include as many triple patterns (TPs) as possible to be processed in a map job. Thus, RDFChain processes as many joins as possible in a single map job.

## 2 Proposed Architecture

RDFChain consists of two components, the data loading component and the query processing one. RDF triples are converted into an N-triple format which is natively supported by Hadoop and then loaded using map jobs. At this stage, we employ the bulk load of HBase instead of directly putting every triple into a table. We also create all the statistics [3] required by our join execution.

### 2.1 Storage Design

The proposed method stores RDF triples as follows:

- RDF triples with the terms that co-exist in both the subject and object parts of triples are located in the $T_{com}$ table. A RDF triple with the term as a subject is considered as a Subject-Predicate-Object (SPO) triple. A RDF triple with the term as an object correspond to Object-Predicate-Subject (OPS).
- SPO triples which are not located in $T_{com}$ are stored in $T_{spo}$.
- OPS triples which are not located in $T_{com}$ are stored in $T_{ops}$.

If a term is used not only as a subject but also as an object in triples, it would be a row key in $T_{com}$. $T_{com}$ has two column-families to represent OPS and SPO schemas. A predicate comes to a column. The subject and object terms become the values of the corresponding columns for OPS and for SPO, respectively. A subject may have several predicates and a predicate may also have several objects. This means that triples are stored in a triple group by a subject as a row. Although $T_{com}$ may be sparse and have a lot of empty fields, empty fields do not occupy storage in HBase. When looking into rows in $T_{com}$, some OPS and SPO triple groups share the same row key.

[2] http://hbase.apache.org.

These triple groups indicate chain pattern relationship. In other words, the target triples of a chain pattern join must exist in $T_{com}$. RDFChain does not index the predicates of RDF triples. Having a table with predicates as row keys has serious scalability problems because the number of predicates in an ontology is usually fixed, relatively small in RDF datasets [3].

### 2.2 Query Planning

A join pattern is determined by the relationship of join variables in a query. Subject-Subject (SS) and Object-Object (OO) relationships are subject to star pattern joins while Object-Subject (OS or SO) relationships are subject to chain pattern joins. A query graph is a directed graph derived from a query. A triple pattern is referred to as a node and each join pattern is referred to as an edge. A logical plan is derived from a query graph. A logical plan includes a set of triple pattern groups (TPGs) and the join order of them. RDFChain determines the logical plan with the selection rules proposed. The selection rules focus on reducing the number of bindings. A chain TPG is a priority for grouping of TPGs.

### 2.3 Query Execution

Where a triple pattern has rdf:type as its predicate, we analyze the class hierarchy in the corresponding ontology tree by utilizing $T_{com}$ instead of storing the triples inferred and then extends the logical plan. The logical plan is transformed into a physical plan for actual join processing on MapReduce and HBase. Each TPG in a logical plan is transformed into a map job in a physical plan. RDFChain makes table mappings for each TPG using an HBase index and accesses tables with an HBase filter based on the structure of a TPG. The first map job uses HBase tables as the query input. The intermediate result of each map job is stored in a distributed file system and then taken as an input of a map job iteration.

For a star pattern join, RDFChain efficiently retrieves a row through a single storage access in a map job since its storage has a triple group by subject or object as a row. For a chain pattern join, RDFChain only scans $T_{com}$ since the triple groups satisfying a chain pattern are located in $T_{com}$. Therefore, $T_{com}$ is more efficient for a chain pattern join due to the reduced number of storage access. In multi-way joins, only the rows with possibility of satisfying a chain pattern join are passed on as inputs of map job iterations, thereby reducing the processing time.

Two chain TPGs with disjoint join variables are compatible [4]. RDFChain estimates the cost of processing compatible sets in a map job. The cost of a map-side join is the sum of all the cost-consuming tasks of a map job. Since we do not need to consider the shuffle and reduce phases of a reduce-side join, we only take account of the time to process a join. In a conventional reduce-side join, a map task simply writes data according to a join key for an actual join in the reduce phase. However, the proposed map task of RDFChain is relatively heavy to iterate both binding and pattern matching. So, it may exceed the execution time assigned to the task. This problem can be resolved by a static method of increasing the timeout or a computing power. However, for a stationary time duration in a map task environment, we split TPGs based

on a threshold to dynamically solve this problem. We limit the number of TPGs which can be processed in a map job using statistics such as the number of objects of frequently used subject-predicate pairs and the number of predicate-object pairs for every subject. We are able to estimate an intermediate result. The number of bindings for join variables dominates the number of map jobs. All map jobs run sequentially. If the cost does not exceed time limit, a single map job is generated.

## 3 Experimental Results

We used a Jena SPARQL parser and the Amazon's Elastic MapReduce service. Ten clusters of large instances were run on Hadoop 0.20.2 and HBase 0.92.0. We experimented with the LUBM 10K, LUBM 20K and LUBM 30K datasets, which consist of 1.3, 2.7 and 4.1 billion triples respectively, and a subset of benchmark queries, Q1, Q2, Q3, Q4, Q7 and Q9 [5]. The queries include simple and complex structures and provide star and chain pattern joins. Q3, Q4, Q7 and Q9 require type inference. We compared RDFChain with HadoopRDF [6] in terms of reduce-side join and MAPSIN [2] for map-side join.

RDFChain showed the best performance in large non-selective queries (Q2 and Q9). In particular, Q2 and Q9 have a complex structure, a low selectivity due to unbound objects, and a relationship of a chain pattern join. RDFChain greatly reduced the size of the intermediate results by limiting RDF triples to actual candidate rows which can satisfy a chain pattern join. RDFChain also shows smaller number of storage accesses than MAPSIN. Since $T_{com}$ is a common subset of $T_{spo}$ and $T_{ops}$, it scales down the scan space. RDFChain splits TPGs with compatible mappings and process the divided TPGs in a map task. So, the number of map jobs decreases in turn.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2013R1A2A2A01016327).

References

1. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A Comparison of Join Algorithms for Log Processing in Mapreduce. In: Proc. International Conference on Management of data (2010)
2. Przyjaciel-Zablocki, M., Schätzle, A., Hornung, T., Dorner, C., Lausen, G.: Cascading Map-Side Joins over Hbase for Scalable Join Processing. CoRR, abs/1206.6293 (2012)
3. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL basic graph pattern optimization using selectivity estimation. In: WWW, pp. 595–604. ACM (2008)
4. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
5. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics 3, 158–182 (2005)
6. Husain, M., McGlothlin, J., Masud, M., Khan, L., Thuraisingham, B.: Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. IEEE Transactions on Knowledge and Data Engineering, vol. 23, no. 9, pp.1312-1327 (2011)