H_2RDF: Adaptive Query Processing on RDF Data in the Cloud

Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos and Nectarios Koziris
Computing Systems Laboratory, School of ECE, National Technical University of Athens
{npapa, ikons, dtsouma, nkoziris}@cslab.ece.ntua.gr

ABSTRACT

In this work we present H_2RDF, a fully distributed RDF store that combines the MapReduce processing framework with a NoSQL distributed data store. Our system features two unique characteristics that enable efficient processing of both simple and multi-join SPARQL queries on virtually unlimited number of triples: Join algorithms that execute joins according to query selectivity to reduce processing; and adaptive choice among centralized and distributed (MapReduce-based) join execution for fast query responses. Our system efficiently answers both simple joins and complex multivariate queries and easily scales to 3 billion triples using a small cluster of 9 worker nodes. H_2RDF outperforms state-of-the-art distributed solutions in multi-join and nonselective queries while achieving comparable performance to centralized solutions in selective queries. In this demonstration we showcase the system's functionality through an interactive GUI. Users will be able to execute predefined or custom-made SPARQL queries on datasets of different sizes, using different join algorithms. Moreover, they can repeat all queries utilizing a different number of cluster resources. Using real-time cluster monitoring and detailed statistics, participants will be able to understand the advantages of different execution schemes versus the input data as well as the scalability properties of H_2RDF over both the data size and the available worker resources.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed Systems

Keywords

RDF, SparQL, Hadoop, MapReduce, HBase, NoSQL

1. INTRODUCTION

With the advent of the Semantic Web, vast opportunities have emerged for semantic stores to better match user needs. A wide variety of RDF (or triple) stores such as Jena SDB [11], BigOWLIM [12], Sesame [2], etc have been designed in order to parse, store and query RDF data that become constantly available.

Research has focused either on optimizing centralized queries or on distributing query processing (especially due to the data explosion [7]): In the first case, the creation of multiple indices that materialize a different number of ⟨s,p,o⟩ index combinations for SPARQL queries has been applied (e.g., [4, 16, 14]). Several approaches that distribute data and/or computation have also been proposed to tackle the big data problem. Representative such systems include Virtuoso Cluster Edition [6], OWLIM Enterprise [12], 4store [10], HadoopRDF [13], etc.

A problem that has not been tackled so far is efficient processing of *both* simple and complex, multi-join SPARQL queries in a distributed fashion. In SPARQL, even a simple query may translate to multiple triple patterns which have to be joined. Centralized systems are highly dependent on main memory constraints in order to process those joins, making them highly vulnerable to the growth of the data size ([9, 3, 15]) and query selectivity. Current distributed systems operate on datasets of virtually any size, but feature poor query execution: Centralized execution for queries with small input and high selectivity is greatly more efficient than distributed processing. To gain optimal performance in both selective and non selective queries, we believe that an adaptive approach is required.

In this work we demonstrate the H_2RDF system, a distributed RDF store that combines a multiple-indexing scheme with BigTable [1] and MapReduce (M/R) [5]. H_2RDF is a high-performance system that allows distributed SPARQL query processing. It creates and distributes three RDF indices on subject, predicate and object over an HBase cluster of commodity nodes. H_2RDF features a join executor module that, for any given join, selects the most advantageous join scenario, choosing between centralized and fully distributed (through the M/R framework). In this demonstration paper, we present the H_2RDF system and make the following contributions:

- We describe how RDF data are stored in a horizontally scalable NoSQL store (HBase). Our system allows bulk-import M/R jobs in order to load, index and keep statistics on large RDF data sets.
- We describe a number of different join strategies which take the query selectivity together with the inherent features of the M/R and HBase systems into account to minimize the processing time.
- We present a query execution engine that, based on a join cost model, chooses the best method on a per-join basis. In effect, this module adaptively chooses the best join algorithm as well as its distributed or centralized execution.

In this demonstration, we will allow participants to interact with H_2RDF on four levels:

WWW 2012 Companion, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1230-1/12/04.

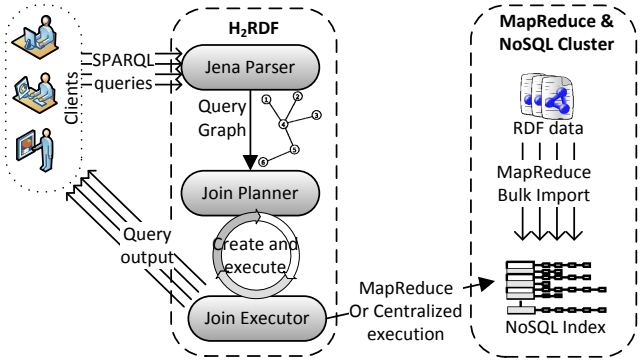

Figure 1: H_2RDF architecture

(1) Dataset used: we allow users to choose dataset from a selection of preloaded LUBM [8] datasets. (2) SPARQL query: users can write their own SPARQL query or select one of LUBM's test queries. (3) Cluster specifications: we allow users to choose the number of concurrent mappers/reducers for query execution. (4) Join algorithm: users can choose the join algorithm used, notice the performance differences and familiarize themselves with our adaptive join execution model.

2. ARCHITECTURE

In Figure 2 we present an overview of H_2RDF's architecture. The system receives RDF triple datasets that are imported into HBase using one M/R, highly efficient bulk import job. The same job also creates all the required statistics needed by our join planner algorithm. Queries are parsed using Jena's SPARQL *parser*, to ensure syntax correctness and create the query graph. The *Join Planner* iterates over the query graph and greedily chooses the join that needs to be executed, according to the selectivity and the cost of all possible joins. Each join is executed by the *Join Executor* module that decides which algorithm (distributed M/R or centralized) will be used for every join. Centralized joins are executed in a single cluster node, while distributed joins launch M/R jobs to process them. Below we describe each module in more detail.

2.1 HBase Indexing

Our goal is the efficient execution of all different SPARQL queries. To achieve that, we materialize three of the six possible indices, namely the *spo, pos* and *osp* combinations. A six index approach can have better performance only for certain queries that contain filters on variables. For all other queries, 3 indices suffice for optimal performance.

Indices are stored in HBase tables in the form of key-value pairs. In this section we describe the H_2RDF *spo* index. The same hold for the other two indices. We use the name SP_O to indicate that we keep a B+ tree based on the combination of subject and predicate values.

The SP_O index is responsible for triple patterns that have either bound subject or bound subject and predicate. The concatenation of subject and predicate values creates the row key, whereas the column identifiers of the current row consist of all the objects associated with the particular subject-predicate combination. All indices store only the 8-byte MD5Hashes of {*s,p,o*} values; a table containing the reverse MD5Hash to value mappings is kept and used during object retrieval. Index statistics (i.e., the number of *objects* for the specific *subject-predicate* combination and the number of *predicate-object* combinations for every *subject*) are kept in special columns and rows of each index table. Subject-predicate bound queries are answered with an exact-key lookup for the rowid that results from the *sp* combination. Subject bound queries are answered with a range query ([subject,`increment`(subject)]).

Subject	Predicate	Object	Index
_	_	_	any
?	_	_	pos
_	?	_	osp
_	_	?	spo
?	?	_	osp
_	?	?	spo
?	_	?	pos
?	?	?	any

Table 1: Index for each query pattern combination.

Table 1 shows the eight different types of triple patterns, corresponding to all combinations of bindings in a triple. For each pattern, the table indicates the index that can be used to retrieve the corresponding data efficiently; "?" denotes the existence of a variable in a triple's position, while "_" means that the position is bound (i.e., fixed). For example, the triple pattern (?, _, _) can be answered using the POS index, as it has bound predicate and object. The patterns having all positions bound, or all unbound, (_, _, _) and (?, ?, ?), can be answered by any index. For (_, _, _), we can select the index having the smallest B^+ tree depth, which is usually OSP. For (?, ?, ?) we choose an index considering any joins that must be performed on the triple pattern.

2.2 Join Execution

A key point in implementing a system capable of evaluating SPARQL queries is determining the way that the system executes joins between triple patterns. Our system is designed to execute both distributed and centralized joins. Distributed joins are executed using MapReduce, while centralized joins are executed in a single cluster node. In this section, we describe the different strategies used to execute distributed and centralized joins.

SPARQL queries with multiple joins are executed by feeding the results of one join to the next. Therefore, we choose to have the same I/O specifications for joins. We store all bindings in the value part of key-value pairs without using the key part. The value part has the following pattern:
$jpat\_var1\$bindings\_var2\$bindings....varN\$bindings$,
where var1 ... N are the different join variables, *bindings* contains one or more values of the corresponding variable and *jpat* is a unique id for each query pattern or join result which helps us recognize the origin of each key-value pair. This format gives some grouping properties, that allows the representation of multiple combinations of bindings in one key-value pair. We are now ready to describe the different strategies used to execute joins in H_2RDF.

Map phase join: The input data of the *Map Phase Join* comes from all joined triple queries formed in key/value pairs of the above format. Mappers read values contained in each pair and break them up to find the join variable. For each join variable binding, they produce a key-value pair with the binding as the key and the bindings for all other variables contained in the input pair as the value. The pattern id is also added in the value. Key-value pairs produced by mappers are sorted and grouped together based on their key.

Reducers take as input for each join variable's binding, a list of values that correspond to it. The join is performed by checking which of the keys were contained in all input queries and by counting the different pattern ids. Reducers create the output by simply merging the key and the corresponding list of values.

Reduce phase join: This algorithm is based on the idea that one of the patterns receives a very small number of input data compared to the rest. Using only this pattern as input, we manage to reduce the amount of data processed and achieve better performance for selective joins. The map function is exactly the same as in the map phase join. The difference is in the reduce phase: We only get the bindings that come from the input query. For every mapped binding, we search our indices to see if it matches with the other queries. This approach, however, is not always the best choice. In joins where all input queries have large input it becomes ineffective because it needs many index accesses.

Partial input join: This algorithm combines the advantages of both Map and Reduce phase joins. It allows the choice of a variable number of input triple patterns and utilizes both preceding join algorithms: Input triple patterns are joined using Map Phase Join, while the rest are joined using Reduce Phase Join. This algorithm allows us to have the best performance in all types of different joins. Naturally, the performance of the algorithm largely depends on whether we make the correct choice for the input pattern(s). Using the statistics gathered during bulk import, we are able to know the exact size of input for every triple query and select the correct input ones.

Centralized join: A M/R job needs a large amount of time for initialization. When input data is small, this initialization time becomes comparable or significantly larger than the required data processing time. In a cluster of 10 nodes, it takes almost 30 seconds for a M/R job, with no input data, to finish. To achieve optimal performance in all cases of different input size joins, we allow the join to be executed in a single node, without launching a MapReduce job. To cooperate with MapReduce joins, the centralized joins use the same input/output format and implement the same partial input algorithm. The choice between distributed or centralized execution is done greedily using an offset parameter that represents the size of data that can be centrally processed during MapReduce's initialization overhead. This is then compared to the cost of a M/R join.

Partial Input Join Cost Model: Joins are executed using our partial input join algorithm. This algorithm is very flexible and can be controlled by its input triple queries (patterns). Let M be the set of input triple patterns and R be the set of the rest of the triple patterns of the join. Thus, M patterns are joined using Map Phase Join while R of them are joined using Reduce Phase Join. We measure the complexity of the M/R job using only read operations and index accesses. The cost of the map and reduce stage is:
$C_{Map} = index + n_1 o_1 read + \cdots + index + n_m o_m read$
$C_{Red} = |n_1 \cap n_2 \cdots \cap n_m| \cdot (index + o_1 read + index + o_r read)$,
where: $index$: time to scan the B+ tree, $read$: time to read a key-value pair, n_i: number of bindings for the joining variable of the i^{th} triple pattern, o_i: number of bindings for the non-joining variables that correspond to one binding of the joining variable (refers to the i^{th} triple pattern), m: number of queries in M and r: number of queries in R.

In the map phase, we only read bindings that match with the M queries. Bindings that match with all the M queries pass the map phase join and their number is $|n_1 \cap n_2 \cdots \cap n_m|$. This number can be approximated by $min\,(n_1, n_2, \ldots, n_m)$. For each of them, we retrieve the bindings that match with the R queries during the reduce phase join. The cost of a partial input join is:

$$Cost_{PIJ} = O\left(\sum_{i \in M} n_i o_i\right) + O\left(\min_{i \in M} n_i \sum_{j \in R} o_j\right)$$

which can be easily computed using the statistics held in our indices.

Table 2: Import and query exec times for LUBM 10K & 20K for H_2RDF, HadoopRDF and RDF-3X

	LUBM10k			LUBM20k		
	H_2RDF	HadRDF	RDF3X	H_2RDF	HadRDF	RDF3X
Imp(min)	118	336	562	282	794	1526
Q1(sec)	0.6	149	0.4	0.6	352	0.4
Q2(sec)	417	838	1158	709	1187	3305
Q3(sec)	0.8	222	0.5	0.9	425	0.6
Q4(sec)	2.1	1245	0.8	2.3	2570	0.8
Q7(sec)	1.9	1809	0.7	2.0	3475	0.8
Q9(sec)	651	1378	260	967	2803	Failed

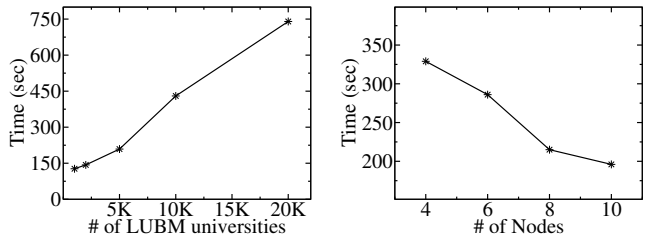

Figure 2: Distributed join execution for different number of universities and nodes for the Q2 query

3. EXPERIMENTS

Our experimental setup consists of a variable number of worker nodes and a single machine in the role of HDFS, MapReduce and HBase master. The worker nodes have 2 Quad-Core E5405 Intel Xeon®CPUs @ 2.00GHz, 8 GB of RAM and a 500GB disk, while the master has similar CPUs and disk, but only 4 GB RAM. Each worker node runs 5 mappers and 5 reducers, each consuming 512MB of RAM. We use Hadoop v. 0.20.205 and HBase v. 0.92.0.

We compare H_2RDF's query and import performance using two current state-of-the-art systems: The RDF-3X [14] centralized store and HadoopRDF [13] distributed system. We utilize the LUBM 10K and LUBM 20K datasets, consisting of 1.3 and 2.7 billion triples respectively with a subset of the benchmark queries. RDF-3X runs on a single worker node while both HadoopRDF and H_2RDF were executed using a cluster of 9 worker and 1 master nodes. Table 2 confirms our premise: Our adaptive execution has the best performance for large, non-selective queries (Q2 and Q9). Regarding selective queries, H_2RDF executes in centralized manner, with directly comparable to the centralized system. In all cases, our approach vastly outperforms the distributed system under comparison.

Figure 2 shows the scalability results for Q2 (6 triple patterns and 3 joining variables) M/R execution. In the first graph, we test the scalability of our system versus different dataset sizes in a 9-node cluster. We clearly note that the M/R execution time is almost linear to the size of input data. In the second graph, we measure the scalability

of the system versus the number of available worker nodes. The tests are executed using the LUBM5k dataset. We notice that the processing is highly scalable since more nodes reduce execution time almost linearly.

4. DEMONSTRATION DESCRIPTION

For demonstrating H_2RDF, we use a comprehensive, real-time GUI that attendees will utilize to interact on dataset, query, cluster size and join execution levels. A sample of the H_2RDF functionality is available as a screencast in `http://www.youtube.com/watch?v=38hsYzsQnAM`.

Dataset Specification: We provide a set of pre-indexed LUBM datasets to avoid the time-consuming loading operation. These datasets vary from 140 million to 3 billion RDF triples with sizes ranging from 2.5 GB to 258 GB. Users will be given an overview of the characteristics of each set they wish to query before actually selecting it. The variety of sizes allows a better understanding of the scalability properties of H_2RDF.

SPARQL Query Specification: Participants will be able to execute different SPARQL queries on the loaded datasets. They can choose to execute one of the documented LUBM test queries: A subset of the total 14 LUBM test queries is selected to provide a good mixture of both simple and complex structures, OWL reasoning, and multiple types of joins. For each of the pre-selected queries, the exact SPARQL form, description and characteristics of the query (i.e., number of triple patterns, number of variables, number of required joins, selectivity) will be shown in order to appropriately comment on the performance of H_2RDF. Additionally, attendees will be able to specify their own SPARQL query using a text-area field.

Cluster Resources: Participants will have the ability to vary the amount of dedicated cluster resources used in order to process each selected or manually created query. Our interface allows three choices on the total number of Mapper and Reducer processes that can be concurrently working on a given query. This way, attendees can observe the effect that increased resources have over the respective speed-up in execution time. Using this feature, we can demonstrate how demanding, high-input join queries scale to the number of cluster workers while joins that can be executed on a single node remain unaffected.

Join algorithms: H_2RDF implements a hybrid join execution model that contains centralized and M/R joins (full input and partial input joins). To discuss on the performance and operation of each algorithm, users will be able to execute each query using one of those algorithms. In this way, participants will observe first-hand why some algorithms are more suitable for specific types of queries than others. Moreover, they will be able to execute the query in "auto"-mode, i.e., using our adaptive approach that selects the optimal algorithm.

The progress of query processing is shown through the Hadoop JobTracker's site, which presents, in real-time, all relevant job metrics. Participants can observe the intermediate outputs of each join and the final query output from the cluster's HDFS site. After query execution, selected parameters and aggregate metrics such as total execution time and output size will be displayed in order to offer direct comparisons and foster discussion among participants.

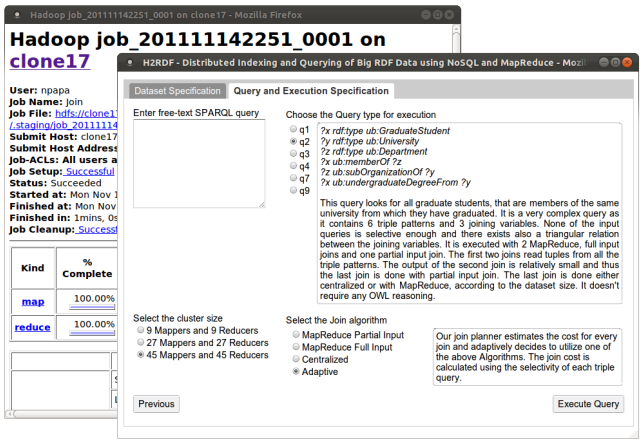

Figure 3: H_2RDF demo interface.

5. ACKNOWLEDGMENTS

This work was partially supported by the European Commission in terms of the ARCOMEM FP7 project (FP7-ICT-2009-6).

6. REFERENCES

[1] HBase Homepage. http://hbase.apache.org.
[2] Sesame. http://www.openrdf.org.
[3] A. Harth et al. Data Summaries for On-Demand Queries over Linked Data. In *WWW '10*, 2010.
[4] D. J. Abadi et al. SW-Store: a Vertically Partitioned DBMS for Semantic Web Data Management. *The VLDB Journal*, 18:385–406, April 2009.
[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[6] O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. *Networked Knowledge-Networked Media*, pages 7–24, 2009.
[7] J. Gantz and D. Reinsel. Extracting Value from Chaos. *IDC research report, Framingham, MA, June*, 19, 2011.
[8] 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*, 2005.
[9] P. Haase, T. Mathäß, and M. Ziller. An Evaluation of Approaches to Federated Query Processing over Linked Data. In *I-SEMANTICS*, 2010.
[10] S. Harris, N. Lamb, and N. Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. In *SSWS*, pages 94–109, 2009.
[11] J. J. Carroll et al. Jena: Implementing the Semantic Web Recommendations. In *WWW*, pages 74–83, 2004.
[12] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM–a Pragmatic Semantic Repository for OWL. In *WISE*, pages 182–192. Springer, 2005.
[13] M. Husain et al. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *TKDE*, 23(9):1312 –1327, Sept. 2011.
[14] T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. *VLDB*, 1:647–659, Aug 2008.
[15] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^2Bench: A SPARQL Performance Benchmark. In *ICDE*, pages 222–233, 2009.
[16] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *VLDB*, pages 1008–1019, 2008.