H_2RDF+: An Efficient Data Management System for Big RDF Graphs

Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras*, Nectarios Koziris
Computing Systems Laboratory, School of ECE, National Technical University of Athens
{npapa, dtsouma, ikons, nkoziris}@cslab.ece.ntua.gr
*Management Science and Information Systems Rutgers University
karras@business.rutgers.edu

ABSTRACT

The proliferation of data in RDF format has resulted in the emergence of a plethora of specialized management systems. While the ability to adapt to the complexity of a SPARQL query – given their inherent diversity – is crucial, current approaches do not scale well when faced with substantially complex, non-selective joins, resulting in exponential growth of execution times. In this demonstration we present H_2RDF+, an RDF store that efficiently performs distributed Merge and Sort-Merge joins using a multiple-index scheme over HBase indexes. Through a greedy planner that incorporates our cost-model, it adaptively commands for either single or multi-machine query execution based on join complexity. In this paper, we present its key scientific contributions and allow participants to interact with an H_2RDF+ deployment over a Cloud infrastructure. Using a web-based GUI we allow users to load different datasets (both real and synthetic), apply any query (custom or predefined) and monitor its execution. By allowing real-time inspection of cluster status, response times and committed resources the audience will evaluate the validity of H_2RDF+'s claims and perform direct comparisons to two other state-of-the-art RDF stores.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Distributed Databases

Keywords

RDF, SparQL, Hadoop, MapReduce, HBase, NoSQL, Joins

1. INTRODUCTION

The unprecedented increase in the production of RDF data is greatly attributed to the advent of the Semantic Web in conjunction with the data deluge of modern times. This reality triggered the emergence of specialized RDF (or *triple*) stores, with a goal of achieving small query response times and efficient storage for arbitrarily large datasets. Consequently, schemes that utilize multiple indexing [11, 16], optimized algorithms for RDF joins [3], pure graph data storage[15, 4], etc. have been employed for this purpose. While many research and commercial tools offer centralized RDF data management (e.g., [5, 2]), data growth dictates distributed storage and indexing (e.g., [8, 17]).

Current distributed triple stores decentralize some or all the stages of RDF data management. Yet, they do not flexibly adjust their behavior with respect to the query in hand, either optimizing a single join algorithm or not elastically allocating platform resources. Yet, SPARQL queries require execution adjustment with respect to both query input and its complexity. Specifically, distributed approaches have not yet taken advantage of maintaining all permutations of RDF elements, namely *spo, pso, pos, ops, osp* and *sop* indexes. Such a scheme can offer the following advantages: (1) All SPARQL triple patterns can be answered efficiently using a single index scan on the corresponding index. For example, a triple pattern with bound subject and variable predicate/object can be answered using a range scan on the *spo* or the *sop* index. (2) Merge-joins that exploit the precomputed orderings can be extensively employed. The existence of all six indexes guarantees that every join between triple patterns can be done using efficient merge joins. More expensive join algorithms are needed only when joining unordered intermediate results.

In this work we demonstrate H_2RDF+ [13], a highly efficient, open-source[1] RDF data management system that maintains both of these properties while moving towards a distributed and scalable environment. We summarize the main contributions of the system as follows:

- H_2RDF+ uses HBase for indexing triples. This scheme allows bulk-import jobs to load and index large RDF datasets. The distributed index is optimized by applying aggressive compression and minimizing the storage requirements. The latter is coupled with the use of an intermediate result materialization that maintains groups of bindings.
- Fully scalable, distributed (MapReduce-based) versions of the well-studied multi-way Merge and Sort-Merge join algorithms are implemented.
- A join cost model and a greedy planner are used to decide, on a per-query basis, on the order of joins and the platform resources (central or distributed) of their execution.

Results have shown that H_2RDF+ can be orders of magnitude faster than the state-of-the-art centralized *RDF-3X*

[1] http://h2rdf.googlecode.com

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2594535.

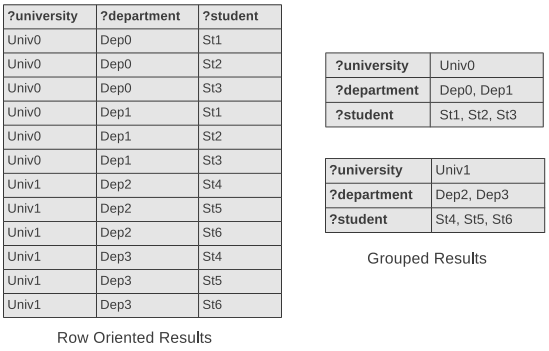

Figure 1: Grouped intermediate results

[11] for complex, non-selective joins, while being only tenths of a second slower in selective ones. Moreover, it proves 6–8 times faster than its previous version [14] and up to orders of magnitude faster than an alternative MapReduce-based scheme [9], scaling easily to 14 billion triples (2.5TB) using a cluster of 35 VMs.

In this demonstration, we will allow participants to interact with a cloud-based deployment of H_2RDF+ controlled via a web UI on three levels: (1) Data Indexing: we allow users to choose a dataset to load and index between synthetic with variable size (LUBM [6]) and real ones (Yago2 [7]). Indexing times and resulting index sizes will be available for inspection by the audience. (2) Querying: users may write their own SPARQL query or select a predefined one to execute. During execution, the audience can inspect the produced plan, the progress and size of intermediate results as well as the cluster resources used. (3) Comparison: the same datasets will be accessible from both *H_2RDF* [14] and *RDF-3X* [11] systems, allowing their direct comparison.

2. H_2RDF+ ARCHITECTURE

H_2RDF+[13] materializes six different RDF indexes, one for each possible *permutation* of subject-predicate-object values; these permutations are *spo, pso, pos, ops, osp* and *sop*. These indexes are stored using HBase tables and allow the retrieval of any simple triple pattern at minimal cost. Compared to the three index approach followed in H_2RDF, the maintenance of all lexicographic RDF indexes allows for: 1) the replacement of H_2RDF's Hash joins with efficient, scalable Merge joins for all joins between indexed triple queries, 2)the use of Sort-Merge joins in cases of joins between both intermediate non-ordered results and RDF index scans. In both cases the network overhead of shuffling data introduced in Hash joins is minimized.

Aggregated index statistics are also materialized and can be used to estimate triple pattern selectivity as well as join output size and join cost. We introduce two categories of aggregated indexes:

- With two out of the three triple elements bound, namely *sp_o, ps_o, po_s, op_s, os_p* and *so_p*. For example, the *sp_o* table contains a set of (subject, predicate, count) records, were the count is the number of triples that contain the respective combination of subject, predicate.
- With one bound element, namely *s_po, p_so, p_os, o_ps, o_sp* and *s_op*. For example, the *p_so* index contains a set of (predicate, count, average) key-values, where count is the number of distinct subjects related to this predicate and average is the average number of objects related to each subject.

Dataset	Raw Size	RDF-3X	H_2RDF	H_2RDF+
LUBM1k	28 GB	9 GB	25 GB	7 GB
LUBM10k	276 GB	77 GB	214 GB	62 GB
LUBM20k	549 GB	156 GB	529 GB	121 GB
Yago2	26 GB	12 GB	33 GB	10 GB

Table 1: Comparison of storage requirements

In addition, we implement a bulk, MapReduce, loading process that can handle the indexing of massive RDF datasets. It consists of 4 highly scalable MapReduce jobs that:

- Translate RDF literals to integer IDs with respect to the literal's occurrence frequency in the dataset. For example, a very frequent predicate will get an ID with value close to zero. Both the String-ID and the ID-String dictionaries are stored in separate HBase tables.
- Generate and load HBase tables for all 6 RDF triple indexes along with their respective aggregated statistics.

H_2RDF+ utilizes an aggressive compression scheme for storing its indexes using: 1)variable length encoding for writing IDs in conjunction with frequency based String-ID mapping, 2)Google Snappy compression [1], also known as "Zippy" compression to further compress the resulting HBase tables. As depicted in Table 1, although storing 6 rather than 3 indexes and more detailed statistics, H_2RDF+ manages to have smaller space requirements than both its previous version and RDF-3X that maintains a similar collection of indexes and aggregated statistics.

Regarding join algorithms, H_2RDF+ implements the well known multi-way Merge join and multi-way Sort-Merge join algorithms, utilizing scalable MapReduce jobs executed over our distributed HBase indexes. The major contribution here is the use of the HBase indexes to generate load balanced total ordered partitions for the distributed execution of joins. The notion of *largest* query triple scan is introduced(i.e., the query scan that spans the most HBase regions). We use its HBase partitioning as a total order partition for our join. Furthermore, Merge joins are executed using Map-only job that process locally the HBase regions of the *largest* scan.

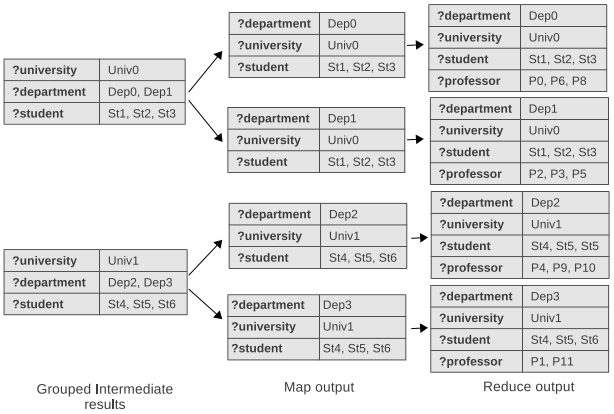

Figure 2: Join on grouped intermediate results

Another, innovative feature of our join algorithms is the use of a lazy materialization of intermediate tuples. By lazy materialization we refer to maintaing groups of intermediate results rather than generating all the combinations of intermediate tuples. Our lazy materialization maintains groups

	Yago2				LUBM10k			LUBM20k			LUBM100k	
	H_2RDF+	H_2RDF	HadoopRDF		H_2RDF+	H_2RDF	HadoopRDF	H_2RDF+	H_2RDF	HadoopRDF	H_2RDF+	H_2RDF
Import(min)	31	26	72	Import(min)	182	168	198	385	312	815	1154	985
YQ1(sec)	0.9	0.9	52	LQ1(sec)	0.6	0.6	152	0.8	0.8	378	0.8	0.9
YQ2(sec)	1.5	1.7	68	LQ3(sec)	0.8	0.8	231	0.9	1	449	1.1	1.1
YQ3(sec)	154	952	1832	LQ4(sec)	2.1	2.4	1289	2.3	2.4	2650	2.4	2.5
YQ4(sec)	87	728	1495	LQ2(sec)	95	635	915	131	880	1367	412	1853
				LQ9(sec)	151	787	1488	292	1034	2933	890	2761

Table 2: Performance comparison of H_2RDF+, H_2RDF and HadoopRDF for LUBM and Yago2 datasets

	Yago2				LUBM10k				LUBM20k			
Resources	8CPU/8GB RAM		64CPU/128GB RAM		8CPU/8GB RAM		64CPU/128GB RAM		8CPU/8GB RAM		64CPU/128GB RAM	
	H_2RDF+	RDF-3X	H_2RDF+	RDF-3X	H_2RDF+	RDF-3X	H_2RDF+	RDF-3X	H_2RDF+	RDF-3X	H_2RDF+	RDF-3X
Import(min)	164	157	26	149	912	605	162	576	2075	1526	349	1398
YQ1/LQ1(sec)	0.9	0.7	0.9	0.7	0.6	0.4	0.6	0.3	0.8	0.4	0.9	0.4
YQ2/LQ4(sec)	1.5	1	1.6	0.9	2.1	0.8	2.1	0.7	2.3	0.8	2.2	0.8
YQ3/LQ2(sec)	241	3037	138	1929	373	2297	89	1277	706	Failed	119	2065
YQ4/LQ9(sec)	123	2973	79	2068	411	68	141	51	753	Failed	264	289

Table 3: Performance comparison of H_2RDF+ and RDF-3X

of bindings that contain: 1) a set of the names of variables contained in the result, 2) for each variable, a list of its bindings. The bindings contained inside a group must satisfy the property of all-to-all connection, i.e., the respective tuples can be materialized by a nested loop over all variables.

As an example, suppose that we execute the following join:

```
?department ub:subOrganizationOf ?university .
?student ub:undergraduateDegreeFrom ?university .
```

Our sorted indexes can retrieve all departments and students grouped per university. We exploit this grouping and instead of generating all row oriented results depicted in Figure 1 we only maintain a group per university. During join execution the groups are split on demand according to the sequence of joins. For example, lets assume that we want to use the above results in the following join:

```
?department ?university ?student .
?professor ub:worksFor ?department .
```

During the map phase of our Sort-Merge join algorithm, the groups are split according to the join variable binding, creating one group for each department. In the reduce phase groups of professors per department are retrieved from the index and are merged with the inputs to form the output groups depicted in Figure 2. We can observe that our lazy materialization process is space efficient avoiding the storage of multiple replicas of bindings.

Finally, H_2RDF+ introduces a greedy planner that decides on both the sequence of joins and their execution utilizing Merge joins, Sort-Merge joins and distributed or centralized adaptive execution. A detailed join cost mode, based on the maintained index statistics and HBase's index scan performance, is also devised in order to allow for accurate join cost estimations. Both centralized and distributed join costs are examined by our greedy planner resulting in plans that can be executed either using MapReduce or a single node of the cluster. This approach provides scalable performance for large joins and interactive responses for selective joins avoiding the MapReduce initialization overhead.

3. EXPERIMENTS

Cluster configuration: The experimental setup consists of an OpenStack private cluster of 6 VM containers. Each container has a 2×6-core Intel Xeon®CPUs at 2.67GHz, 48 GB of RAM and two 2TB disks setup with RAID 0. Worker VMs feature a 2-virtual core processor, 4GB of RAM and 300GB of storage space, allowing the cluster to support a total of 36 VMs. The clusters we use for our evaluation consist of 25 worker VMs plus a single VM in the role of the HDFS, MapReduce and HBase master. Each worker VM runs 2 mappers and 2 reducers, each consuming 512MB of RAM. We utilized Hadoop v1.1.2 and HBase v0.94.5 respectively.

Compared Systems: We compare the performance of H_2RDF+ against three state-of-the-art RDF stores: RDF-3X [11], HadoopRDF [9] as well as the first version of our distributed system H_2RDF [14]. We evaluate the latest version (v0.3.7) of RDF-3X [11, 12]. HadoopRDF was built using the latest SVN rev. 158 from the project repository.

Data Sets Used: We utilize two datasets in our evaluation. The Yago2 dataset [7] consists of real data gathered from various resources such as Wikipedia, WordNet, GeoNames, etc, and contains more than 120 million triples. The LUBM dataset generator [6] creates datasets with academic domain information. By varying LUBM's number of *universities* between 1K to 100K, we create datasets ranging from 140 million (25GB) to 13.8 billion triples (2.5TB).

In order to provide a direct comparison among the different systems, we first test the performance of H_2RDF+ versus the other distributed systems. Table 2 registers the data import times and response times for the selected SPARQL queries using LUBM10k, LUBM20k, LUBM100k and Yago2 datasets. For a fair comparison to the centralized RDF-3X, we run both systems using the same *total* amount of resources: For RDF-3X, we use two single-server configurations (a 2×Quad-Core with 8 GB RAM, 8 GB swap and 1TB disk and a 4×16-Core with 128 GB RAM and 1TB disk); for H_2RDF+, we break the corresponding server in VMs and use them as our cluster infrastructure. These results are reported in Table 3.

We divide SPARQL queries in two categories: 1) selective, small queries and 2) non-selective, complex queries. H_2RDF+ performs noticeably better in large non-selective queries: it proves almost 7× better than H_2RDF and 10× better than HadoopRDF. We also outperform RDF-3X in most of the complex queries both when running on the small and the large server configuration, due to RDF-3X's memory requirements and single threaded execution engine. For small, selective queries H_2RDF+ uses centralized execution and manages to obtain performance comparable to RDF-3X. The difference in performance is mainly attributed to the local disk B+Trees used by RDF-3X that offer better seek and scan performance than our distributed HBase indexes. We also note that HadoopRDF has really poor performance for all selective queries due to the fact that it only executes MapReduce joins that process all input data and cannot take advantage of query selectivity.

To sum up, H_2RDF+ manages to correctly identify selective vs. non-selective queries, performing either distributed or centralized joins. It proves almost as efficient as RDF-3X, with a small difference (few tenths of a second), for selective queries. For more data-intensive queries, H_2RDF+ proves greatly superior to both central solutions and competitive Hadoop-based schemes due to both our join strategy and lazy intermediate result materialization.

4. DEMONSTRATION DESCRIPTION

For demonstrating our system, we use a comprehensive, real-time GUI that attendees will utilize to interact over dataset, query and comparison levels. The UI controls a cloud-based H_2RDF+ deployment over several virtual machines from the ∼Okeanos IaaS [10].

Dataset Specification: Participants will be given the choice to index raw triple-sets of various types and sizes. We provide: (i) A set of LUBM synthetic datasets that vary from 140 million to 3 billion RDF triples (sizes ranging from 2.5 GB to 500 GB) and (ii) the Yago2 dataset consisting of real life RDF data. Users will be given an overview of the characteristics of each set before loading it; after this operation finishes, statistics over the duration and resulting index size will be available, allowing commentary on the efficacy of the MapReduce based load job and the size of the indexes versus compression.

Query Specification: Participants will be able to execute different SPARQL queries on the loaded datasets. Regardless the dataset, they can specify their own query using a text-area field. For synthetic datasets, a list of documented LUBM test queries will be available, selected as to provide a good mixture of both simple and complex structures, OWL reasoning, and multiple types of joins. For each of the 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. Upon proceeding with the execution, the generated execution plan will be shown to the users; real-time progress will be available through the Hadoop JobTracker's site, which presents all relevant job metrics as well as the committed cloud resources. Participants will be able to 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.

Direct Comparison: Possibly the most interesting part of the demonstration is the direct comparison of different RDF engines that can foster discussion among participants. Our UI will offer interactive querying experience with two directly comparable RDF stores, RDF-3X [11] and H_2RDF+'s predecessor [14]. RDF-3X will be deployed over a central, RAM- and storage-rich server and H_2RDF over the same cluster as our new system. Allowing users to perform identical queries over three different stores and analyzing (both real time and aggregate) results enables direct comparisons and, most importantly, conclusions on join, indexing and planning algorithm efficiency. We plan to demonstrate the superior performance of 6-ple indexing that allows efficient Merge-Joins as well as the performance gains brought forth by result grouping and data compression. The different query types that best demonstrate these differences will be pointed out to the audience, that can interactively validate these findings.

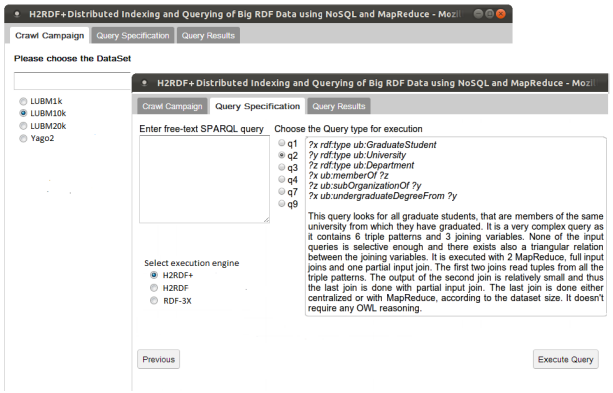

Figure 3: H_2RDF+ demo interface.

Acknowledgment

The research leading to these results has received funding from the European Union, Seventh Framework Programme (FP7/2007- 2013) under grant agreement no 619706.

5. REFERENCES

[1] Google Snappy. https://code.google.com/p/snappy.

[2] Sesame. http://www.openrdf.org.

[3] M. Atre, J. Srinivasan, and J. Hendler. BitMat: A Main-memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries. In *ISWC*, 2008.

[4] V. Bonstrom, A. Hinze, and H. Schweppe. Storing rdf as a graph. In *Web Congress, 2003.*

[5] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *WWW*, 2004.

[6] 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.

[7] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum. YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages. WWW, 2011.

[8] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 2011.

[9] M. Husain, J. McGlothlin, M. Masud, L. Khan, and B. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *TKDE*, 2011.

[10] V. Koukis, C. Venetsanopoulos, and N. Koziris. Okeanos: Building a Cloud, Cluster by Cluster. *IEEE Internet Computing*, 17(3):67–71, 2013.

[11] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDBJ*, 2010.

[12] T. Neumann and G. Weikum. x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases. *VLDB*, 2010.

[13] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs. In *IEEE International Conference on Big Data*, 2013.

[14] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. *H_2RDF*: Adaptive Query Processing on RDF Data in the Cloud. In *WWW*, 2012.

[15] O. Udrea, A. Pugliese, and V. Subrahmanian. GRIN: A graph based RDF index. In *National Conference on Artificial Intelligence*, 2007.

[16] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB*, 2008.

[17] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB*, 2013.