Scalable Indexing and Adaptive Querying of RDF Data in the cloud

Nikolaos Papailiou
Computing Systems Laboratory, National Technical University of Athens
npapa@cslab.ece.ntua.gr

Dimitrios Tsoumakos
Department of Informatics, Ionian University
dtsouma@ionio.gr

Ioannis Konstantinou
Computing Systems Laboratory, National Technical University of Athens
ikons@cslab.ece.ntua.gr

Panagiotis Karras
Management Science and Information Systems Rutgers University
karras@business.rutgers.edu

Nectarios Koziris
Computing Systems Laboratory, National Technical University of Athens
nkoziris@cslab.ece.ntua.gr

ABSTRACT

Efficient RDF data management systems are central to the vision of the Semantic Web. The enormous increase in both user and machine generated content dictates for scalable solutions in triple data stores. Current systems manage to decentralize some or all the stages of RDF data management, scaling to arbitrarily large numbers of triples. Yet, these systems prove highly inflexible in adjusting their behavior relative to the query in hand. Queries over triple data include multiple joins with varying degrees of selectivity and cost. In many cases, a join performed on a single centralized computer node is highly preferable. Thus, both informed query planning and adaptive join execution are necessary to gain optimal performance in both selective and non selective queries. Towards that direction, we describe H_2RDF+, an RDF store that efficiently performs distributed joins over a multiple index scheme. H_2RDF+ materializes 6 RDF indexes and detailed statistics using HBase. In this work, we emphasize on our novel, scalable and efficient MapReduce indexing process that allows H_2RDF+ to handle arbitrarily large RDF datasets. Aggressive byte-level compression is also extensively used to reduce the storage space requirements of the system. H_2RDF+ can also adaptively process both complex and selective queries by adaptively choosing the amount of resources allocated for each join, based on join complexity estimated through index statistics.

1. INTRODUCTION

The Resource Description Framework (RDF) [5] has been proposed for information representation and exchange in the context of the Semantic Web [6]. The Semantic Web vision realization requires efficient RDF data management systems. Several centralized RDF triple stores (e.g., [12, 15, 7], etc) have been proposed, with subsequent research focusing on creating efficient indexing structures for query processing [8, 33, 24]. Such approaches materialize a different number of index combinations that allow for significantly reduced response times. However, centralized solutions are still vulnerable to the growth of the data size [17, 19, 31]. In response, some of the aforementioned systems proposed distributed solutions (e.g., Virtuoso Cluster Edition, Jena Clustered TDB), which, together with new approaches (e.g., [18, 22]) aim to bring forth the desired scalability.

It is evident that, when the data size keeps increasing, distributed approaches can achieve better results. Nevertheless, in order to avoid bottlenecks and fully harness a cluster's power, the distribution must be performed throughout the data management life-cycle: from RDF indexing up till SPARQL query answering. Balance between distributed and centralized processing is required in order to achieve the desired scalability. Indeed, for queries with small input and high selectivity, centralized processing is much more efficient than distributed processing. Yet, current execution engines do not flexibly adjust their behavior with respect to the query in hand. This entails two important aspects: 1) adjusting the way of performing joins with respect to join's complexity, and 2) utilizing only the amount of resources required for each join. Scalability has to be achieved not only with respect to data size, but also with respect to query complexity. Many RDF queries require a large number of joins over multiple variables. Increasing the number of joins should not lead to an exponential growth of response times due to suboptimal join execution order and unnecessary data transfers. Moreover, as queries are not executed in isolation but concurrently by multiple users over a cluster, resource allotment should be fine-grained and match query complexity to maximize query throughput and cluster utilization.

We argue that, in order to gain best-case performance with queries of varying complexity and size and take full advantage of cluster resources, an *adaptive*, *elastic* approach is called for. In this paper we present H_2RDF+, a fully-

SWIM 2014, June 27, 2014, Snowbird, UT, USA.

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

distributed, open-source [1] RDF datastore with full SPARQL querying functionality. Its strength lies in its adaptive query planner and execution engine that adjusts the join execution order, the type of execution (using what join algorithm), the distribution (single- or many-machine via MapReduce) and the amount of resources (map-reduce tasks or threads) for each query. The aforementioned decisions are taken on a per join basis and assisted by a detailed cost-model that analyzes the running time of the join algorithms based on their input, execution type and number of committed resources [26]. The main contributions of this paper are as follows:

- We devise an indexing scheme for storing RDF data implemented in HBase [3], an open-source implementation of Bigtable [13], which allows bulk-import of MapReduce [14] jobs to load and index large RDF data. We thoroughly present our highly-efficient and scalable bulk indexing procedure which is able to create all 6 permutations of RDF indexes along with 12 aggregated indexes used for statistics. Our system is able to index and query RDF datasets with more than 14 billion triples using a cluster of commodity nodes.
- We study the adaptivity and elasticity properties of our query execution engine. Our system decides, on the fly, on the exact amount of resources allocated for each join. If the join is performed over multiple nodes, this corresponds to the number of mapper/reducer tasks. In the centralized execution case, it regulates the number of deployed threads. In both cases, our system maximizes the inherent parallelism found in our join algorithms.

2. RELATED WORK

The question of how to efficiently store and query RDF data has shaped an active research area. Initially, RDBMs were used as RDF triple stores, directly storing all triples in a single table. This approach presented serious scalability deficiencies, which Abadi et al. attempted to rectify [8], proposing to create a separate table for each predicate. The rationale behind this approach is that, as the number of distinct predicates is usually small, the maintenance cost of this approach is not prohibitive. However, this scheme is still unsuitable for queries with unbound predicate, and its performance can deteriorate as input data grows [33]. Thereafter, research has made significant efforts towards efficient RDF indexing and querying. We distinguish existing systems in two categories: centralized engines, and distributed solutions that utilize a cluster for managing large RDF datasets. While distributed approaches have great performance on non selective queries, they are outperformed in selective queries by efficient centralized systems.

2.1 Centralized Systems

Hexastore [33] is one of the most prominent RDF indexing solutions. It materializes six different indices, one for each possible *permutation* of subject-predicate-object values; these permutations are spo, pso, pos, ops, osp and sop. For instance, the spo index contains a list of predicates for each subject, while each predicate p in the list points to a table that contains all objects associated with s by p. The availability of these indices allows for the retrieval of any triple query pattern at minimal cost. The existence of all possible orderings of triples also enables the extensive use of efficient merge joins. A similar approach is followed in RDF-3X [24, 25] along with query optimization strategies. RDF-3X employs six *lexicographic* indices, similar in spirit to the ones in Hexastore, as well as additional aggregated indices that collect statistical information for pairs (instead of triples) of entities and for alone-standing entities, amounting to a total of 15 indices. RDF-3X is used as the state-of-the-art prototype centralized system for querying RDF data.

BitMat [10], proposed a new RDF indexing mechanism that uses the notion of a 3-dimensional $\langle$*subject, predicate, object*$\rangle$ bit matrix. Each matrix element is a bit denoting the presence or absence of the corresponding triple. This 3-d bit matrix is flattened to 2-d matrices creating multiple indices for all possible combinations of subject-predicate-object. A novel query execution approach is also employed. It uses efficient vector-matrix operations to perform a semi-join based, query execution. However, this approach is effective only in a main-memory environment and due to its semi-join nature cannot handle complex graph query patterns that contain cycles.

Other frequently-used, efficient centralized systems include Virtuoso [15], Jena [12] and OWLIM [23]. Still, all aforementioned approaches run on a single machine, limiting their storage and processing capacity.

2.2 Distributed Systems

The vast increase of publicly available RDF data has directed research towards distributed RDF data management systems. A first attempt in this direction, 4store [18], distributes a single POS index over the nodes of a cluster, and employs distributed join algorithms to execute SRARQL queries. However, apart from the deficiency ensuing from having a single index, 4store does not adapt its performance for multiple join queries of various selectivity.

An elementary attempt to utilize HBase and MapReduce for RDF data management was made in [32]. However, the work in [32] does not examine the question of adapting the execution strategy to a centralized or distributed one depending on the nature of the workload. In addition, the system is not aware of query pattern selectivity and thus relies only on joins that process large amounts of data even for selective queries. Furthermore, this approach does not exploit the power of MapReduce to perform multi-way joins and its experimental study is limited to small datasets.

Subsequently, [22] proposed HadoopRDF, a Hadoop-based RDF storage system; HadoopRDF uses HDFS files named after predicate values to partition the input RDF data, thereby creating a POS index; this is not a fully functional POS index, as it can only retrieve subject-object combinations for a given predicate, but not, e.g., subjects for a given predicate-object combination. HadoopRDF performs SPARQL joins in the MapReduce framework, employing an algorithm that greedily reduces the total number of remaining MapReduce joins at each step. Remarkably, this greedy planner does *not* take into consideration the join's selectivity. Finally, joins are executed only with MapReduce jobs inducing large overheads for selective queries.

Two recent works propose NoSQL indexing for RDF data [29, 30]. Both of them use a similar NoSQL indexing scheme stored in HBase[3] for the first and in Accumulo[1] for the later. In [29], MAPSIN, a selectivity aware MapReduce join algorithm is proposed to handle SPARQL joins. This algorithm reads input only from the selective pattern of the join and uses HBase API in order to evaluate all remaining pat-

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

terns. In [30], a centralized join approach is implemented using Accumulo's batch scanners. Our system uses both selectivity aware MapReduce and centralized join algorithms. As we show in our experimental evaluation, no single algorithm is better for all kinds of joins. Thus, a decision module is required in order to achieve best case performance for all types of joins.

H_2RDF [27] uses a three-index scheme and depends on the *Partial Input Hash-join*. This algorithm exploits HBase indexing and checks whether the join contains small input patterns. If this is the case, only those are read from the indexes during the map phase. The remaining patterns are joined using `get` operations on the reduce phase of the join. H_2RDF also uses adaptive centralized and distributed execution. The main differences with H_2RDF+, can be found in the join algorithms, the number of maintained indexes (3 vs. 6), the more detailed statistics, the type and size of IDs and the result grouping used in H_2RDF+ during query execution [26]. Furthermore, H_2RDF offers adaptivity and elasticity properties [9]. It adaptively chooses between distributed and centralized execution as well as the amount of computing resources dedicated for the join processing. It also allows the elastic allocation and deallocation of computing resources according to the query workload. As shown in [9] H_2RDF can scale its computing resources to adapt to the SPARQL query throughput required by the given workload.

An alternative proposal is presented by Huang et al. [21]; this method starts out by partitioning the RDF graph into distinct subgraphs, each stored in a single node running a local RDF-3X instance. Moreover, in a replication scheme, each node keeps information on the graph contents within n *hops* from the contents it owns; this provision allows for unobstructed parallel processing of SPARQL queries satisfying an n *hop guarantee*. In case this guarantee is not satisfied, Hadoop is invoked for distributed join processing. The proposed system suffers from the following drawbacks:

- It utilizes *all* cluster resources for *every* query; thus, even highly selective queries are executed on the entire cluster, although a small subset may contain all the relevant data.
- Very slow import: apart from the centralized graph processing it also needs a large amount of time to transfer data to the corresponding nodes, and load them in individual RDF-3X instances.
- Last, the replication to satisfy the n-hop guarantee amounts to replicating data of size *exponential* in n.

Zeng et al. [34] introduce Trinity.RDF a distributed, in-memory RDF system. They propose a query execution model based on graph exploration that can be viewed as a sequence of semi-joins similar to the approach followed in BitMat. The main drawback of this system is that its performance is bound by the main memory capacity of the cluster, as the whole set of triples needs to be loaded in main memory. Yet, this is not the most scalable approach, especially when a cluster is created by commodity nodes. Moreover, it is the case that local semi-join results are gathered on a central node responsible to produce the final results. This server can be the bottleneck of the query execution when: 1) Handling query graphs that contain cycles. The semi-join query execution engine employed cannot fully reduce the result size for these graphs [11], thus overloading the last step of the execution. 2) The query output is really large the last server will need to generate and write the whole output. This process is limited by the sequential iteration over the result set and the large write I/O requirements.

3. SYSTEM ARCHITECTURE

Figure 1 presents an overview of the H_2RDF+ [26, 28] architecture. The system uses HBase [3] as a distributed indexing substrate for large triple datasets. RDF data is imported in HBase through a bulk import process. Users are able to pose SPARQL queries parsed by a Jena [12] parser that checks the query syntax and creates the query graph. Our *Join Planner* and *Executor* modules choose the join that needs to be executed, the algorithm to be used, the method of execution (centralized or distributed) and the required resources on a per-join basis.

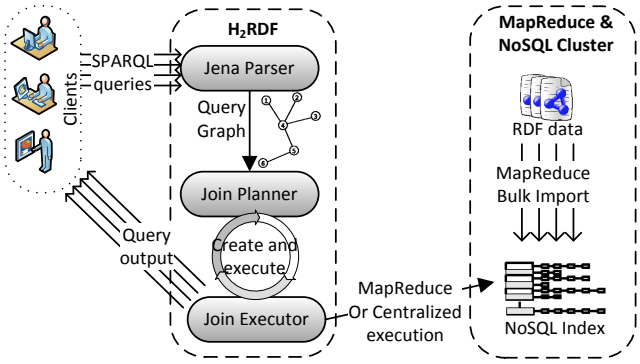

Figure 1: H_2RDF+ architecture

In this section, we explain the decisions made about the number and type of indexes used in our system. Although most state-of-the-art centralized RDF stores use combinations of Hexastore-like indexes, distributed approaches have not yet taken advantage of this technique to offer increased scalability and fast querying. Maintaining all six permutations of RDF elements, namely *spo, pso, pos, ops, osp* and *sop*, offers the following advantages: (1) All SPARQL triple query 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. As stated in [33], 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 the H_2RDF+ case, we maintain both of these properties while moving towards a distributed and scalable environment. We move from local disk B^+-trees to distributed key-value tables (HBase) and from centralized to distributed, MapReduce-based, join processing. Our system is available as an open-source project and offers RDF data indexing and SPARQL querying functionality as well as a user friendly web-interface.

4. INDEXING RDF DATA

H_2RDF+[26] materializes all six permutations of subject-predicate-object values of RDF triples; these permutations are *spo, pso, pos, ops, osp* and *sop*. To be able to provide efficient access to those indexes in a distributed environment we store them using HBase tables and thus achieve the desired index scan and search capabilities. 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.

H_2RDF+ also features a detailed join cost model that is required in order to estimate the execution cost of different joins and decide the amount of computing resources that need to be dedicated for the processing of each join. To facilitate the join cost estimation 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, where 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.

4.1 Bulk RDF data indexing using MapReduce

In this section, we thoroughly describe our MapReduce bulk indexing process that can handle the indexing of massive RDF datasets. It consists of four highly scalable MapReduce jobs that:

- Translate RDF literals to integer IDs with respect to the literal's occurrence frequency in the dataset. 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. The frequency aware ID mapping in conjunction with our variable length encoding scheme for writing IDs inside our indexes achieves great reductions in storage space requirements.
- Generate and load HBase tables for all 6 RDF triple indexes along with their respective aggregated statistics.

In order to handle web scale RDF datasets our bulk indexing process needs to minimize the number of I/O and network operations and avoid unnecessary iterations over the RDF dataset. It also avoids the execution of HBase API calls for each tuple insertion; instead, bulk import MapReduce jobs directly create HFiles (the HBase file format) which are then loaded directly in HBase tables.

4.1.1 First MapReduce job

In the first MapReduce job each mapper reads one block of RDF triples and generates a sorted-map that contains all the unique string labels found in the file followed by a counter that represents the number of times each string label was found in the respective RDF block. At the cleanup phase of each mapper this sorted-map is used to produce the following information:

- For each RDF block, a file that contains all its distinct string labels is created. This file will be used in subsequent steps in order to efficiently retrieve the relevant IDs that are needed to translate all the triples in the block.
- Each mapper emits all the wordcounts of the sorted-map in order for the reducers to produce a global string label wordcount.
- We also use a sampling rate in order to sample the input triples and generate balanced partitions on both the distinct string label space and the indexing space (for all possible triple orderings). Concerning the distinct string label space for each sampled triple the mappers emit three special key-values one for each of its string labels(s, p, o). All the sample key-values are sent to a special reducer that is responsible for generating a load balanced partitioning of the string label space. At this moment, we cannot yet create the partitioning of the indexing space because we require the translated IDs that are not yet decided. Therefore, we just generate for each RDF block a sample file that contains its sampled triples.

The first job issues two types of reducers: 1)the first reducer which handles the sampled key-values and generates the string-label partition and 2)the wordcount reducers that sum up all local counters for each distinct label. Each wordcount reducer maintains a sorted list containing its wordcount key-values sorted by their counts. The reducers write their output at the cleanup phase and thus generate blocks of locally ordered, according to the count, key-values.

4.1.2 Second MapReduce job

The second job is responsible for giving globally unique and frequency aware IDs to all distinct string labels present in the dataset. The mappers of this job read the locally ordered, according to the count, wordcount output produced in the previous step. In order to avoid globally sorting all the distinct string labels according to their frequency count we assign IDs using a loose global order that requires no more communication information. The first MapReduce job utilized a HashPartitioner to split the labels between the reducers and therefore we can assume, with high probability, that all output blocks will contain labels from all the frequent and non-frequent classes of string labels. Taking advantage of this property we assign IDs using the following formula:

$$ID = \begin{cases} locID * R + redID, & locID < min \\ offset[redID] + locID - min, & locID >= min \end{cases}$$

where:
ID : is the global ID assigned to a label
$locID$: is the local ID inside each block that is assigned according to the local order of counts
min : is the minimum number of key-values produced by a wordcount reducer in the first MapReduce job
$redID$: is the ID of the reducer that produced the respective output block
$offset[]$: all reducer IDs will be interleaved until the minimum number of keys is reached. In order to avoid introducing holes to the assigned ID space we then start assigning contiguous ID regions to each reducer. This is achieved using the $offset$ table that contains the first ID of the respective contiguous region. The $offset$ is computed using the values $numKeys[redID] - min$ for each reducer.

Both the min number of output key-values and the $offset$ table can be easily computed by the output of the previous job. We can observe that this formula interleaves IDs between the wordcount blocks, generates a loose order of as-

signed global IDs and introduces no holes to the assigned ID space. We note here that by generating this loose ordering we avoid resorting and reshuffling the data and introducing unnecessary overhead.

Using the above formula, the mappers of the second job can assign independently the global IDs for each string label. For each string label the mappers emit two key-values in order to create both the string-to-ID and ID-to-string HBase dictionaries. This job also takes as input the distinct string label blocks generated in the first step. The mappers that read those files emit for each distinct label a key-value containing as key the label and as value the block ID.

To handle the two indexing spaces we use two separate partitioners for this job. The first partitioner is the string label partitioner computed in the previous step. The second partitioner handles the ID space and just splits it in continuous regions of a certain size. The load balancing of the ID space partitioner is achieved due to the fact that we used a contiguous ID space that contains no holes. We use two types of reducers for this job. The ID space reducers simply create the respective HFiles and directly load them to HBase tables. The string label reducers both generate the respective HFiles and also a file that contains for each string-ID pair a list of blocks that this is required. The second file is used in the following job to translate the distinct string label blocks.

4.1.3 Third MapReduce job

The third job handles the translation of the distinct string label blocks. It utilizes the files generated in the previous job. We assign one reducer to each RDF triple block. The job reads the files that contain for each string-ID pair a list of blocks that this is required and generates for each a key-value with key the block ID and value the string-ID mapping. Each reducer gets all the translations of an RDF triple block and just outputs them to an HDFS file.

4.1.4 Fourth MapReduce job

The last job parses the RDF triples again. First, each mapper reads the translation file for the corresponding RDF block and loads it into a memory hash map. It then parses the RDF triples, translates the string values and emits key-values for all 6 different orderings of the RDF triple. This job also requires the computation of a load balanced total order partitioner for the hexastore indexing space. Before starting the job, we translate the sample triple files created in the first job using our HBase dictionaries and generate a load-balanced partitioner for the indexing space. Each reducer of this job takes as input a sorted range partition of the indexing space and, while iterating over it, computes the aggregated statistics described above and creates the corresponding HFiles for both the primary and the aggregated indexes. The statistics maintained in the aggregated indexes, described in the previous section, can be easily computed while iterating over the sorted indexes and thus are computed without introducing additional network or I/O overhead.

4.1.5 Indexing storage space

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 [2], also known as "Zippy" compression to further compress the resulting HBase tables. Our variable length encoding scheme is presented in Table 1 and can support IDs with up to 8 byte length.

Positive Prefix	Negative Prefix	Total Bytes	ID Bits
10******	01******	1	6+0=6
110*****	001*****	2	5+8=13
1110****	0001****	3	4+16=20
11110***	00001***	4	3+24=27
111110**	000001**	5	2+32=34
11111100	00000011	6	0+40=40
11111101	00000010	7	0+48=48
11111110	00000001	8	0+56=56
11111111	00000000	9	0+64=64

Table 1: Variable length encoding scheme

Our encoding scheme uses variable length prefixes in order to encode the length of each ID. The specific selection of prefixes achieves the following objectives:

- Maintains the byte ordering property of the encoded IDs. This means that a raw byte comparator would order the variable length IDs in the same order as a value comparator. This property is really important because our indexes depend heavily on byte order.
- The variable length prefixes allow more IDs to be encoded with less bytes. A static prefix definition would require at least 5 bits to encode all the different cases. This means that only 2^3=8 IDs could be encoded using only one byte. However we can encode 2^6=64 IDs using only one byte. The same is true for all IDs that can be encoded with less than 5 bytes.

5. ADAPTIVE QUERY EXECUTION

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. Both the join execution methods and their join cost model used in H_2RDF+ are thoroughly presented in [26]. In this section, we focus on the resource adaptivity properties offered by our system. H_2RDF+ is able to decide, on the fly, on the number of resources required to process each join in hand. In effect, it is able to automatically estimate the amount of required resources, be they threads (centralized case) or map/reduce tasks (distributed case), on a per-join basis. The adaptive decisions for each join are done during runtime and they scale according to the estimation of the join cost.

For small join costs we use centralized execution and scale the resources using a different number of concurrent threads. Both our merge and sort-merge join algorithms can be executed in parallel by partitioning the join variable key space. Utilizing the statistics held in our aggregated indexes we can estimate the number of join variable bindings contained in each of the joined relations. We use the estimation of the maximum number of bindings contained in a join relation to decide at runtime how many threads will be launched.

We then greedily split the join variable key space and assign work to different execution threads. We launch a thread only if it is estimated to process more than a minimum amount of input bindings. We also pose a limit to the number of concurrent threads in order to avoid costly context switching between them.

Larger joins are executed using distributed MapReduce jobs. The resources required for the MapReduce execution also scale with the join cost. The resources available on a MapReduce cluster are the number of concurrent mappers and reducers. Assuming these are set for a specific cluster, we want to occupy only the number of mappers and reducers required for the execution of each join. The cost of a MapReduce join is proportional to its input data. As discussed in [26], input data are split in HBase regions, each region having a configured size. In our implementation, every map task handles one HBase region and thus the region size is configured to contain the amount of data required to amortize the initialization overhead of launching the task. If the region had less data, the initialization overhead of a map task would be greater than the actual processing of the data. Therefore, the number of resources occupied is proportional to the size of input and, by extension, to the join cost. If the map tasks launched by the join are less than the cluster's concurrent mappers (which we anticipate to be the case for less costly joins on medium to large size clusters), the remaining resources will be proportionally allotted to other users' joins. In the opposite case, all the mapper slots and the cluster resources will be occupied.

6. EXPERIMENTS

In this section we present a performance evaluation of the H_2RDF+ system.

Cluster configuration: Our 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 variable numbers of VMs (10 to 35) 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 [24], HadoopRDF [22] as well as the first version of our distributed system H_2RDF [27]. We evaluate version 0.3.7 of the centralized *RDF-3X* system. HadoopRDF was built from source using SVN rev. 158 from the project repository.

Data Sets Used: To test the system under web-scale, realistic conditions we utilize two datasets. The Yago2 dataset [20] consists of real data gathered from various resources such as Wikipedia, WordNet, GeoNames, etc, and contains more than 120 million triples. This dataset is relatively small; we use it to show that distributed query execution can perform better even for small datasets when large non-selective queries are required. The LUBM dataset generator [16] creates datasets with academic domain information, enabling a variable number of triples by controlling the number of *university* entities. By varying this parameter between 1K to 100K, we create datasets ranging from 1.4 million (25GB) to 13.8 billion triples (2.5TB). This dataset is widely used to compare performance of triple stores especially when arbitrarily large datasets are required. Lehigh university has also published a suite of test queries [4] that offer a good mixture of SPARQL queries.

6.1 Indexing storage space requirements

As mentioned in section 4, H_2RDF+ uses aggressive compression to reduce the storage space required for storing all 6 hexastore indexes along with aggregated statistics indexes in HBase. Table 2 registers the storage requirements of different RDF storage systems for the LUBM [16] and Yago2 [20] datasets.

Dataset	Raw Size	RDF-3X	H_2RDF	H_2RDF+	H_2RDF+ (Snappy)
LUBM1k	28 GB	9 GB	25 GB	27 GB	7 GB
LUBM10k	276 GB	77 GB	214 GB	241 GB	62 GB
LUBM20k	549 GB	135 GB	529 GB	545 GB	121 GB
Yago2	26 GB	12 GB	33 GB	35 GB	10 GB

Table 2: Comparison of storage requirements

The "Raw Size" column contains the size of the dataset serialized using the *N-Triples* format. Although storing 6 rather than 3 indexes and more detailed statistics, H_2RDF+ manages to have smaller space requirements than its previous version due to: 1) the smaller ID values, as H_2RDF uses the 8-byte MD5-hash of the string values as ID, 2) the byte-level variable length encoding in conjunction with the frequency-aware ID mapping, 3) the block level Snappy compression. RDF-3X also offers a highly compressed storage scheme due to its gap compression [24] (stores only the difference between subsequent triples in the index). The difference between the storage requirements of RDF-3X and H_2RDF+ results mainly from the frequency-aware ID mapping and the block-level Snappy compression used in H_2RDF+.

6.2 Bulk RDF indexing

In this section we evaluate the performance of the H_2RDF+ bulk indexing process. As described in section 4 H_2RDF+ indexing process is consisted of 4 MapReduce jobs that materialize all 6 combinations of RDF indexes. To test the efficiency of our indexing method we compare it using RDF-3X, HadoopRDF as well as the first version of our distributed system H_2RDF.

We first test the scalability properties of all the compared systems regarding the RDF dataset size. To do so we utilize the LUBM dataset generator that can generate RDF datasets with variable size. We import LUBM datasets containing 1K to 20K universities, i.e., 0.14 to 2.7 billion triples (28 to 549GB of data respectively). H_2RDF+, H_2RDF and HadoopRDF were executed using a cluster of 25 worker and 1 master nodes, while RDF-3X uses a 4×16-Core server with 128 GB RAM and 1TB disk. Total import times are presented in Figure 2. This is the time needed for all systems to load the full dataset according to their indexing scheme.

RDF-3X, being a centralized system, parses all triples sequentially in order to create its indexes. It doesn't exploit the parallelism capabilities offered by the modern multicore architecture of CPUs. It also reads the input data several times in order to generate the various different orderings of the triples. This iterative scan of input data results in an increasing import complexity. As we can see in Figure 2,

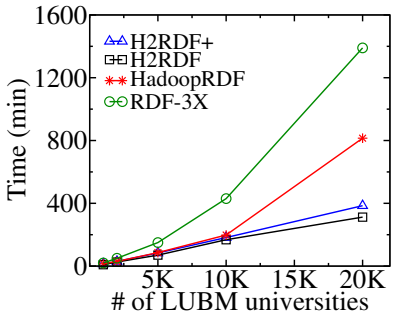

Figure 2: Import scalability versus dataset size

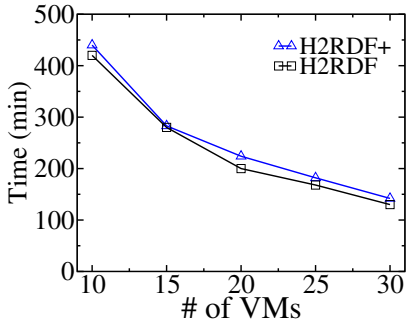

Figure 3: Import scalability versus cluster resources

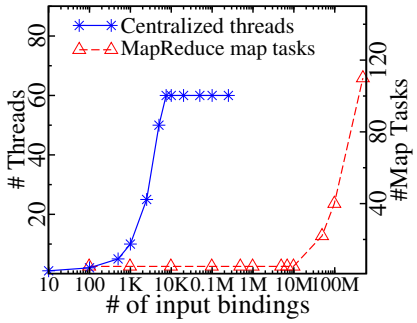

Figure 4: Adaptive query resource allocation

RDF-3X introduces the slowest, among the compared systems, import times for loading RDF datasets.

HadoopRDF needs to execute four different MapReduce jobs which take as input the whole dataset. This means that it needs to scan the data four times resulting in low import performance. Additionally, some of these jobs do not equally partition the reduce input data and thus overload some reducers while leaving others idle. The load balance between the available computing resources is one of the most important properties that need to be handled in order for distributed systems to offer good scalability properties. We can observe that HadoopRDF, while materializing only one ordering of the triples in raw HDFS files, requires 2× more time than H_2RDF+ for loading the LUBM20k dataset.

We also compare H_2RDF+ to our previous H_2RDF system. H_2RDF used 2 MapReduce jobs to materialize 3 of the 6 RDF triple indexes. The first job was a sampling job that created a load balanced partitioning of the indexing space, while the second one used the partitioning to generate and load the 3 materialized HBase RDF indexes. In Figure 2 we can observe that H_2RDF+ manages to import 3 additional indexes and keep more detailed statistics than H_2RDF at a mere 10-20% overhead. This is mainly attributed to:

- Our optimized indexing procedure that minimizes the times that the raw dataset is read. While requiring 4 MapReduce jobs to import the dataset, only 2 of them read the raw dataset while the rest process output files that are quite smaller than the original dataset. This greatly reduces the I/O time needed for executing our import process.
- The aggressive compression used in all the indexing steps. We use our variable length encoding to write all intermediate and final results of our indexing process and thus save both storage space and I/O time.
- The extensive use of sampling to generate load balanced partitions for all our MapReduce computation steps.

Another important point is the indexing scalability to the number of available computing resources. We import the LUBM10k dataset (1.3 billion triples) using clusters with different number of worker nodes. We use clusters with 10, 15, 20, 25, 30 worker nodes. The corresponding results are presented in Figure 3. We can observe that H_2RDF+ manages to maintain the scalability properties of H_2RDF while introducing a more complex RDF indexing process. Using the MapReduce framework we can gain almost linear speedup when we increase the number of worker nodes: At 10 workers we achieve an import speed of 49 Ktriples/sec, while using 30 nodes we almost triple the import speed at 142 Ktriples/sec. We also observe that H_2RDF+ introduces only a small time overhead (10-20%) compared to H_2RDF in all tests.

6.3 Join Planner and Elastic Execution

In this section, we compare the performance of our join algorithms and test our planner's decisions over different input queries. Moreover, we demonstrate the adaptive execution properties of our system. In order to test the scalability of our algorithms we generate the following benchmark setup: We use a cluster of 25 VMs and the `ud:takesCourse` property from the LUBM20k dataset which contains 515 million triples that describe connections between students and courses. We randomly sample the corresponding data using variable sampling rates and store the sampled triples in a new HBase index. Figure 4 shows the ammount of computing resources occupied by our join algorithms to execute a merge join of the full `ud:takesCourse` relation of the LUBM1k dataset(25 million triples) with the sampled ones using different join algorithms. We range the sampled triples from 10 to 500 million.

Our merge join algorithm can be executed either in a centralized or a distributed environment. In Figure 4 we depict the number of dedicated resources for every join execution. We can see the amount of resources committed to the join scale proportional to the join cost. For centralized joins, the planner scales the number of concurrent threads while for MapReduce joins it scales the number of mappers. For our cluster configuration, threads were launched only when they had a minimum amount of 100 binding to process. We also set the maximum amount of concurrent threads to 60.

Regarding distributed joins our join planner can decide on the fly for the amount of map tasks required to execute the join according to its cost. This is done by finding the maximum join input scan as mentioned in section 5, i.e. the join relation scan that spans the most HBase regions. When finding the maximum scan, one map task is assigned to each of its regions. As the size of the sampled triples range from 10 to 500 million triples the size of the largest scan varies. When the sampled triples are less than the 25 million of the `ud:takesCourse` relation the maximum scan is the `ud:takesCourse` relation of LUBM1k that spans 4 regions and thus 4 map tasks are launched. When the sampled triples get bigger they become the larger scan and then on the utilized map tasks scale proportionally to their size.

7. CONCLUSIONS

In this paper we presented H_2RDF+, a fully distributed RDF store capable of storing and querying arbitrarily large

amounts of triples. We thoroughly presented the indexing MapReduce procedure used to materialize all 6 permutations of RDF triples using HBase tables. Our experiments show that the presented indexing process is scalable with respect to both the dataset size and the number of available computing resources. Our system is able to load large scale RDF datasets using a cluster of computing resources and achieves an indexing throughput of 142 Ktriples/sec while utilizing a cluster of 30 small-sized VM nodes. We have also presented the aggressive compression used in our NoSQL, HBase indexes. Another contribution of H_2RDF+ is its adaptive query execution engine. The amount of computing resources dedicated to the execution of a query are shown to be proportional to the query cost and range from centralized threads to distributed map tasks. This property allows H_2RDF+ to occupy, in a per-join basis, only the required computing resources and leave the rest cluster resources available to process multiple concurrent queries.

Acknowledgment

This research has received funding from IKY fellowships of excellence for postgraduate studies in Greece - SIEMENS program.

8. REFERENCES

[1] Accumulo. http://accumulo.apache.org.

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

[3] HBase. http://hbase.apache.org.

[4] LUBM queries. http://swat.cse.lehigh.edu/projects/lubm.

[5] Resource Description Framework (RDF). http://www.w3.org/RDF/.

[6] Semantic Web. http://www.w3.org/standards/semanticweb/.

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

[8] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: a Vertically Partitioned DBMS for Semantic Web Data Management. *VLDBJ*, 2009.

[9] E. Angelou, N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. Automatic scaling of selective SPARQL joins using the TIRAMOLA system. In *SWIM*, 2012.

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

[11] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *JACM*, 1981.

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

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. BigTable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Comm. of the ACM*, 2008.

[15] O. Erling and I. Mikhailov. Virtuoso: RDF Support in a Native RDBMS. In *Semantic Web Information Management*, pages 501–519. Springer, 2009.

[16] 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.

[17] P. Haase, T. Mathäß, and M. Ziller. An Evaluation of Approaches to Federated Query Processing over Linked Data. In *I-SEMANTICS*, 2010.

[18] S. Harris, N. Lamb, and N. Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. SSWS 2009.

[19] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich. Data Summaries for On-demand Queries over Linked Data. In *WWW*, 2010.

[20] 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.

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

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

[23] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM - A Pragmatic Semantic Repository for OWL. In *WISE 2005*.

[24] T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. *PVLDB*, 2008.

[25] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.

[26] 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.

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

[28] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H2RDF+: An Efficient Data Management System for Big RDF Graphs. In *SIGMOD 2014*. ACM.

[29] M. Przyjaciel-Zablocki, A. Schätzle, T. Hornung, C. Dorner, and G. Lausen. Cascading map-side joins over HBase for scalable join processing. 2012.

[30] R. Punnoose, A. Crainiceanu, and D. Rapp. Rya: a scalable RDF triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, 2012.

[31] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^2Bench: A SPARQL Performance Benchmark. In *ICDE*, 2009.

[32] J. Sun and Q. Jin. Scalable RDF Store Based on HBase and MapReduce. In *ICACTE*, 2010.

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

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