H_2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs

Nikolaos Papailiou [*#], Ioannis Konstantinou [*#], Dimitrios Tsoumakos [‡#], Panagiotis Karras [$], Nectarios Koziris [*#]

[#]*ATHENA Research and Innovation Center,* [*]*Computing Systems Laboratory, National Technical University of Athens*
{npapa, ikons, nkoziris}@cslab.ece.ntua.gr

[‡]*Department of Informatics, Ionian University*
dtsouma@ionio.gr

[$]*Management Science and Information Systems Rutgers University*
karras@business.rutgers.edu

***Abstract*—The proliferation of data in RDF format calls for efficient and scalable solutions for their management. While scalability in the era of big data is a hard requirement, modern systems fail to adapt based on the complexity of the query. Current approaches do not scale well when faced with substantially complex, non-selective joins, resulting in exponential growth of execution times. In this work we present H_2RDF+, an RDF store that efficiently performs distributed Merge and Sort-Merge joins over a multiple index scheme. H_2RDF+ is highly scalable, utilizing distributed MapReduce processing and HBase indexes. Utilizing aggressive byte-level compression and result grouping over fast scans, it can process both complex and selective join queries in a highly efficient manner. Furthermore, it adaptively chooses for either single- or multi-machine execution based on join complexity estimated through index statistics. Our extensive evaluation demonstrates that H_2RDF+ efficiently answers non-selective joins an order of magnitude faster than both current state-of-the-art distributed *and* centralized stores, while being only tenths of a second slower in simple queries, scaling linearly to the amount of available resources.**

Keywords*—*RDF, SPARQL, MapReduce, HBase, Distributed Indexing, Distributed Merge-Joins

I. Introduction

The Resource Description Framework (RDF) [4] has been proposed for information representation and exchange in the context of the Semantic Web [5]. RDF data is stored in the form of $\langle s(ubject), p(redicate), o(bject)\rangle$ triples, in which a relationship between subject and object is stated using the predicate. SPARQL [7], the standard language for querying RDF data, allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns.

Several centralized RDF triple stores (e.g., [12], [13], [6], etc) have been proposed, with subsequent research focusing on creating efficient indexing structures for query processing (e.g., [8], [26], [21]). Such approaches materialize a different number of index combinations that allow for significantly reduced response times. Still, centralized solutions are vulnerable to the growth of the data size [16], [24]. In response, distribution of existing schemes and novel distributed ones (e.g., [15], [19]) aim to bring forth the desired scalability.

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 committing on a specific join algorithm or the execution platform's resources. SPARQL queries often require multiple joins over a (possibly large) number of triple patterns and variables that the query contains. Thus, a resolution engine would need to adjust its execution with respect to both query input and complexity. Single joins range in complexity as input and selectivity range. As the number of joins and intermediate results to be processed increase, this should, correspondingly, not lead to an exponential growth of response times.

Distributed approaches have not yet taken advantage of maintaining all permutations of RDF elements, namely *spo, pso, pos, ops, osp* and *sop* indexes [26]. Such a scheme offers 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 merge joins. More expensive join algorithms are needed only when joining unordered intermediate results. In the H_2RDF+[1] case, we maintain both of these properties while moving towards a distributed and scalable environment. We summarize the main contributions of this work as follows:

- We devise an indexing scheme for storing RDF data implemented in HBase [2], which allows bulk-import jobs to load and index large RDF datasets. We optimize the retrieval capabilities of our distributed index 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. Our indexes compensate the fact of being nearly $10\times$ slower than disk-based B^+trees by achieving great scalability and parallel scanning performance.
- We present fully scalable, distributed (MapReduce based) versions of the well-studied multi-way Merge and Sort-Merge join algorithms.
- We devise a join cost model and use the estimated cost of each join to greedily decide on the order of joins and the platform (central or distributed) of their execution.
- We perform a thorough experimental evaluation of our system. Results show that H_2RDF+ can be orders of magnitude faster than a state-of-the-art centralized store [21] for complex, non-selective joins, while being only tenths

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

of a second slower in selective ones. Moreover, it proves 6–8 times faster than its previous version [23] and up to orders of magnitude faster than an alternative MapReduce-based scheme [19]. H_2RDF+ easily scales to 14 billion triples (2.5TB) using a cluster of 35 VMs, providing linear scalability in terms of the amount of available resources.

II. Related Work

We present some of the most relevant RDF indexing and querying systems, distinguishing them in two categories: centralized and distributed systems.

Centralized Systems: Hexastore [26] is a centralized solution that materializes six different indexes, one for each possible *permutation* of subject-predicate-object values; these permutations are *spo, pso, pos, ops, osp* and *sop*. The *spo* index, for instance, 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*. These indexes allow the retrieval of any simple triple pattern at minimal cost.

A similar approach is followed in RDF-3X [21] along with query optimization strategies. RDF-3X employs six *lexicographic* indexes (similar to [26]) as well as additional indexes that collect statistical information for pairs and stand-alone entities, amounting to a total of 15 indices. It extensively uses merge joins in order to achieve good performance. However, query execution highly depends on the amount of main memory required to perform joins, presenting problems with joins with small selectivity and large input. The use of single threaded query execution limits RDF-3X's scalability on modern multi-core server architectures. RDF-3X is regarded as a state-of-the-art solution in centralized RDF data stores.

In BitMat [9], RDF triples are indexed via a 3-dimensional $\langle s, p, o \rangle$ bit matrix. Each matrix element is a bit denoting the presence or absence of the corresponding triple. This matrix is flattened to 2-d matrices creating multiple indexes for all possible combinations of subject-predicate-object. However, this approach is effective only in a main-memory environment.

Other frequently-used, efficient centralized systems include Virtuoso [13], Jena [12] and OWLIM [20]. Still, all aforementioned approaches run on a single machine, limiting their storage and processing capacity.

Distributed Systems: In order to tackle the big-data challenge, research has recently moved onward to distributed RDF data management systems. A first attempt in this direction, 4store [15], 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.

HadoopRDF [19] uses Hadoop Distributed File System (HDFS) files named after predicate values to partition the input RDF data, thereby creating a *pos* index. It is not a fully functional index though, as it can only retrieve subject-object combinations for a given predicate, but not, for instance, 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 selectivity. Finally, joins are executed only with MapReduce jobs, inducing large overheads for selective queries.

Efforts have also been made towards optimizing distributed joins using MapReduce [11]. In this work, the authors compare different algorithms for joining big log tables, stored in raw HDFS files. The main difference with H_2RDF+ is that we index our data using HBase. This means that we always process only the amount of data required for each join without having to process the whole dataset. The join algorithms presented in [11] do not take into account any data preprocessing and indexing. We also use a multi-way join scheme that differs from the two-way joins implemented in [11].

H_2RDF [23] 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 (three versus six), the more detailed statistics and the type and size of IDs

An alternative proposal is presented by Huang et al. [18]; 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:(1) Slow import: apart from the centralized graph processing, it also needs a large amount of time to load the corresponding data to individual RDF-3X instances.(2) Its MapReduce joins implement a non optimized, 2-way hash-join scheme.(3) The n-hop guarantee requires size of replication data *exponential* to n.

Zeng et al. [27] introduce Trinity.RDF, a distributed, in-memory 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. This is not a scalable approach, especially given that clusters are comprised by commodity nodes. Moreover, local semi-join results are gathered at a central node responsible for producing 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 [10], thus overloading the last step of the execution. 2) The query output is really large. In this case 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.

III. H_2RDF+ System

A. Indexing Scheme

1) HBase Indexes: HBase is a distributed, NoSQL key-value store that can handle large amounts of data using

commodity machines. HBase tables are, in practice, range partitioned sorted key-value maps. In our system, we use an HBase table for each index. As HBase uses a key-value model, our indexes store all triples in keys and leave the values empty.

RDF triples contain long string URIs and literals that can add a lot of space overhead, especially in the case of multiple indexes. To achieve a space-efficient implementation, we use IDs instead of strings and keep two separate HBase tables that work as dictionaries to translate string values to IDs and vice versa. This mapping from string-based IDs to byte-based IDs is created during data import with respect to the occurrence frequencies of the string literals in the dataset: A very frequent predicate will get an ID with value close to zero. In order to take advantage of the frequency related IDs we apply byte-level, variable-length encoding when storing IDs in HBase. Variable length encoding leads to smaller byte representations for frequently used values and thus achieves high compression.

2) Index Statistics: Apart from the six indexes described above, we keep aggregated index statistics that can be used to estimate triple pattern selectivity as well as join output size and join cost. We have two categories of aggregated indexes:
i) 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) key-values, were the count represents the number of triples that contain the respective combination of subject, predicate.
ii) 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.

3) MapReduce Bulk Import: In order to handle web scale RDF datasets we use a bulk loading process that avoids 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. Our import procedure consists of four highly scalable MapReduce jobs:
The first job operates similarly to a wordcount MapReduce job but also creates the following: 1) for each block of RDF triples, it creates a file that contains the distinct string values that are present inside the block and 2) it samples input triples and creates balanced partitions on both the distinct string value space and the indexing space (for all possible triple orderings).
The second job gives (byte) IDs to string values according to their word-counts and loads both the string-to-ID and ID-to-string HBase dictionaries. This job uses the partitioning on the distinct string value space computed in the first job in order to achieve load balancing.
The third job translates the distinct string value blocks and creates for each block of RDF triples a file that contains the mapping between string values and IDs.
The last job parses again the RDF triples. 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 maps all different orderings using the indexing space partitioning computed in the first job. Each reducer 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 all the indexes.

B. Join Execution Algorithms

Our work makes a twofold contribution relative to the join execution engine: We present a multi-way merge join algorithm and a sort-merge join algorithm, both executed over our distributed index. The former performs efficient joins over already sorted data (i.e., the HBase index tables); the latter performs joins when some of the data is unsorted (i.e., when intermediate results exist). The two algorithms can be executed in both distributed (via MapReduce) and centralized (over a single cluster node) mode.

1) MapReduce Merge Join Algorithm: This algorithm is designed to join multiple triple query patterns over the same variable. For example, suppose that we want to perform the following join on variable `department`:
?person ub:memberOf **?department** .
?department ub:subOrganizationOf ?university .
?department rdf:type ub:Department .
We can get the triples ordered by `department` if we do the following three range scans: (for each range scan we specify the index table and the bound values in the respective order) {pos, ub:memberOf}, {pso, ub:subOrganizationOf}, {pos, rdf:type, ub:Department}. To execute the distributed merge join over those scans, we first specify the largest scan (i.e., the scan that spans the most HBase regions). We implement the merge join algorithm as a Map-only job over the regions of the largest scan. Each mapper processes a sorted partition of the scan (region), which translates to a sorted partition over the join variable's keys. The mapper has a local scanner over the large pattern and initializes the respective scanners over the other query patterns respecting the range of the the join variable's keys.

For example, let us assume that the largest pattern of the above join is the first containing two regions with the following join variable ranges: `[Dep0, Dep5)` and `[Dep5, Dep10)`. Note that we use string values here for readability; the partitions are in the ID space. The first mapper will initialize two scanners: {pso, ub:subOrganizationOf, `[Dep0, Dep5)`}, {pos, rdf:type, ub:Department, `[Dep0, Dep5)`} and merge join them with the local region scanner. The second mapper will handle the range `[Dep5, Dep10)` respectively.

2) MapReduce Sort-Merge Join Algorithm: This algorithm is only used when we join intermediate (thus unordered) results. It can take as input one or more intermediate results and one or more triple queries. For example, suppose that we want to perform the following join on variable `department`:
?y **?department** ?w . (1)
?z **?department** . (2)
?person ub:memberOf **?department** . (3)
?department rdf:type ub:Department . (4)
The first two patterns present intermediate results that contain bindings for all the variables depicted in the pattern's name. These patterns are not ordered by the join variable. At first, we check the triple query scans (triple patterns (3) and (4)) and find the maximum partition of the join variable in the same way that we described above. The sort-merge join is executed as a MapReduce job that takes as input only the intermediate result patterns (triple patterns (1) and (2)). Each mapper reads bindings from the intermediate results and maps them using as key the binding of the join variable. The job uses the maximum join variable partition to produce a global

ordering of the reduce keys. This means that each reducer will get a sorted range of the join variable's keys. The reducer initializes the index scans for its respective key range and then merges all intermediate and triple patterns by iterating over the sorted input. In case we need to join only intermediate results we utilize a hash partitioner and perform a hash join.

For example, let us assume that the largest pattern of the above join is, as before: `[Dep0, Dep5)`, `[Dep5, Dep10)`. The first reducer will get *all* the intermediate bindings in the first range and will initialize two scanners: {pos, ub:memberOf, `[Dep0, Dep5)`} and {pos, rdf:type, ub:Department, `[Dep0, Dep5)`}. The reducer will iterate over all patterns and produce the join results. The same will happen with the second reducer over the second range.

3) Centralized Join Algorithms: We also implement the classic versions of the merge and sort-merge join algorithms in a centralized environment. The only difference is that we use HBase scanners in order to iterate over the sorted relations rather than local B^+-tree or file scanners.

4) Intermediate results format: SPARQL queries involve multiple joins and feeding results of one join to the next. Intermediate results can become really large and grow exponentially with each subsequent join. This is why we need to have a space-efficient representation of the intermediate results. Standard row oriented databases create all result tuples at the end of each join. Instead, we opt for a lazy materialization of intermediate tuples and try to maintain grouped results as much as possible. Our lazy materialization maintains groups 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 need to exploit this grouping as much as possible in order to avoid generating all intermediate result tuples. Assume our database contains 2 universities, each having 2 departments and 3 students. The row-oriented results of the join are depicted in Fig. 1 (left). Instead of materializing all these combinations we store grouped results as depicted in Fig. 1 (right). Note that there is no explicit connection between students and departments (students and departments connect only with the university and not with each other), thus the all-to-all connection property applies. Extending our example with larger figures, if our database contains 100 universities, each of them with 30 departments on average and 100K students, a row-oriented scheme would create $100 \times 30 \times 100K = 300M$ result tuples by replicating a lot of times the IDs of universities and departments. To store these results, we would need to write three times as many IDs ($900M$). For the same example, our scheme would create 100 groups, one for each university, each group containing 30 bindings for the department variable and 100K bindings for the student variable. Thus, we would need to output $100+100\times30+100\times100K$ = 10,003,100 IDs which is orders of magnitude smaller that the previous requirement. We also apply byte level, variable length encoding on IDs and achieve a highly compressed output size.

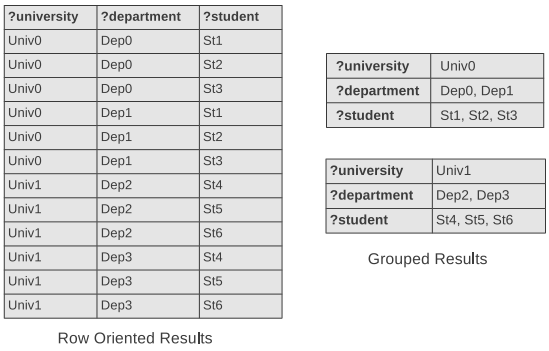


Fig. 1. Grouped intermediate results

As stated before, 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** .
This join, on variable department, is executed using the sort merge join algorithm described in the previous section. In Fig. 2 we can see how we use the grouped results in the join procedure. Initially, in the map phase, we split the group according to the join variable, thus we create one group for each department. Note that the map output is not split across the student bindings because those bindings maintain the all-to-all connection with the rest bindings. 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.

C. Query Planning and Execution

Deciding on the query execution plan is an important aspect that greatly influences performance, since SPARQL queries usually require multiple joins on different variables. The H_2RDF+ planner decides on the execution order of the different joins so as to minimize the total query execution time. To find the optimal join order we have to consider the different combinations in which the joins can be performed. Obviously, the number of choices grows exponentially to the number of joining variables, making the problem computationally expensive. Instead, we use a greedy, cost-based, online planner that decides on the join that must be executed in every step of the query. To derive the costs of possible joins we devise a detailed join cost model that takes advantage of

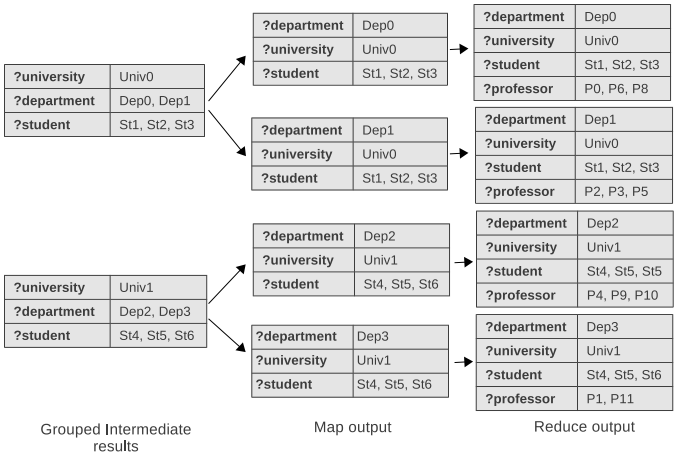


Fig. 2. Join on grouped intermediate results

our stored statistics. Our cost model can be also used to help the planner decide on whether the join will be executed in a centralized or a distributed fashion. The incentive behind this decision is that distributed MapReduce jobs cannot offer real-time response times for small joins and are beneficial only in case of large joins. More sophisticated approaches, like dynamic programming planners[21], can be also applied but this work is beyond the scope of this paper. In this section we present the join cost model as well as our greedy join planner.

1) HBase scan performance evaluation: Our join execution heavily depends on HBase scans and thus in order to derive an applied cost model we need to stress-test their performance. The key parameters of a scan are the seek latency and the read throughput. After doing some experiments on scanning our indexes we found out that a seek operation takes on average 16ms and the average read throughput reaches 400,000 key-values(triples)/second. Detailed performance evaluation for those features can be found in Section IV-A. These values are infrastructure specific and can change across different installations but they can be estimated by a simple benchmarking test that runs once for every different installation.

We integrate this performance knowledge into our merge join algorithm in order to make it more efficient. Except from sequentially scanning the input relations a merge join algorithm may need to jump forward on one relation if we know that there are no possible join results in this range. In this case we need to take the decision of whether to seek to the next position by initializing a new scanner or read all intermediate values sequentially. From the above metrics we can easily note that the time needed for a seek operation is equal to the time needed to sequentially read nearly 6,400 key-values. Thus the merge join algorithm uses the seek operation only if it is expected to discard more than 6,400 key-values.

2) Merge join cost model: The merge join algorithm is controlled by its input triple queries(Q). The total cost of the join (in terms of completion time) is:

$$MJcost(Q) = \sum_{i \in Q} ReadKeys(Q, i)/thr \quad (1)$$

$$ReadKeys(Q, i) = \min\{(\min_{j \in Q} n_j) \cdot o_i \cdot SeekOverhead, n_i o_i\} \quad (2)$$

n_i: number of join variable's bindings for the i^{th} query.
o_i: average bindings of the non-joining variables corresponding to one join variable binding. Refers to the i^{th} query.
thr: the scan throughput discussed earlier.
SeekOverhead: the seek overhead (6,400 key-values)
ReadKeys(Q,i): the number of key-values that will be read from the i^{th} query.

The cost of the merge join algorithm depends on the number of key-values that need to be read. To estimate this size we first find the minimum number of input join keys among the joining queries. A merge join algorithm would need to read at most that amount of keys from each relation using seeks to pass over irrelevant keys. As stated before we use an heuristic to decide whether to perform a seek operation and thus in the worst case scenario our merge join algorithm would always seek paying each time the *SeekOverhead*.

3) Sort-Merge join cost model: In this algorithm we have to join both a set of input scans(Q) and a set of intermediate results(I). The total cost of the join (in terms of time) is:

Algorithm 1 H_2RDF+ PLANNER

```
1: V ← {v_1, v_1, ..., v_n} //join variables
2: TQ ← {tq_1, tq_1, ..., tq_m} //triple queries
3: //TQ(v) triple queries that contain variable v
4: while V ≠ empty do
5:    Jstruct ← empty //Join's required information
6:    v_join ← min_{v_i ∈ V}{Greedy(v_i, TQ(v_i))}
7:    Jstruct.addJoin(v_join, TQ(v_join))
8:    V.remove(v_join)
9:    TQ.remove(TQ(v_join))
10:   if Jstruct.executionType() = MR then
11:      executeMapReduce(Jstruct)
12:   else if Jstruct.executionType() = Cent then
13:      executeCentralized(Jstruct)
14:   end if
15: end while
```

$$SMJcost(Q, I) = (2\sum_{i \in I} n_i o_i + \sum_{i \in Q} ReadKeys(Q \cup I, i))/thr \quad (3)$$

The cost of the sort-merge join algorithm is divided in two main parts. The first part is the cost of joining the intermediate results. The intermediate patterns are read twice, once in the map and once in the reduce phase. For the triple queries we use the same estimation described in the above section.

4) Join Planner: The cost model described in the previous section is a step towards finding the optimal join execution plan, i.e., the join order with the minimum total execution cost. Our planner uses a greedy algorithm that in each step of the execution selects the smallest cost join to be executed.

Our greedy join planner is presented in Algorithm 1. Set V contains all the variables that need to be joined in order to answer the query. Set TQ contains all the triple queries that need to be joined. While V contains more variables we need to execute more joins. Using our greedy function we select the most beneficial variable to be joined. The selected variable is fully joined in the current job (multi-way join), which means that all its queries are joined and we remove it from V.

Our greedy function is presented in Algorithm 2. This function checks if the join requires a merge or a sort-merge join algorithm and then computes the costs of executing the join in centralized or distributed manner. The centralized cost is the cost described in the previous section. The distributed MapReduce cost is computed by dividing the centralized cost by the minimum of partitions and number of mappers in the cluster. This number is the maximum amount of parallelism that will be present when executing the distributed job. We also add an overhead called *MRoverhead* which is the amount of time required to setup a MapReduce job. A MapReduce job with no input data needs at least 30 sec to finish. Thus our incentive is to use centralized jobs when quick responce times can be achieved and leverage the parallelism of distributed execution only when we face large joins.

IV. EXPERIMENTAL RESULTS

In this section we present a thorough 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

Algorithm 2 $Greedy(v, TQ)$

```
1: //TQ contains the triple queries to be joined
2: //Split TQ in scans and intermediate results
3: (Q, I) ← splitPatterns(TQ)
4: if I ≠ empty then
5:     //Sort-merge join
6:     cost ← SMJcost(Q, I)
7: else
8:     //Merge join
9:     cost ← MJcost(Q)
10: end if
11: //Compute MapReduce Cost
12: MRcost ← cost/ min(patitions, mappers) + MRoverhead
13: if cost < MRcost then
14:     Jstruct.addExecutionType(Cent)
15:     return cost
16: else
17:     Jstruct.addExecutionType(MR)
18:     return MRcost
19: end if
```

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 [21], HadoopRDF [19] as well as the first version of our distributed system H_2RDF [23]. We evaluate the latest version (v0.3.7) of RDF-3X [21], [22]. HadoopRDF was built using the latest SVN rev. 158 from the project repository.

All the above systems process queries using dictionary IDs rather than strings and URIs. We have observed that the last step of translating query result IDs to strings is a challenging task for all compared triple stores. In some cases, it requires time comparable or even larger than the actual processing. In this paper, we focus on the join execution engine. Thus, in order to provide a fair comparison, we have also removed the translation task from all the compared systems.

Data Sets Used: To test the system under web-scale, realistic conditions we utilize two datasets. The Yago2 dataset [17] 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 [14] 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 [3] that offer a good mixture of SPARQL queries.

A. Index comparison

In this section we evaluate the performance of our indexing scheme. Initially, we consider space requirements. As mentioned in Section III-A1, H_2RDF+ uses an aggressive compression scheme using variable length encoding and smaller IDs for frequent string values. We also compress our index tables using the Google Snappy compression [1], also known as "Zippy" compression. We choose the Snappy library because it offers very high decompression speed and reasonable compression. Snappy's CPU-efficient decompression algorithm makes it a perfect candidate for NoSQL stores by exploiting the trade-off between I/O and CPU bandwidth.

TABLE I. COMPARISON OF STORAGE REQUIREMENTS

Dataset	Raw Size	RDF-3X	H_2RDF	H_2RDF+ (no Snappy)	H_2RDF+
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	156 GB	529 GB	545 GB	121 GB
Yago2	26 GB	12 GB	33 GB	35 GB	10 GB

In Table I we register the storage requirements of the compared systems for the LUBM and Yago datasets. 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 athe latests 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 [21] (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+ (achieves $\sim$70% storage reduction).

We also study the retrieval efficiency of the indexes and their respective technologies. As mentioned in Section III-C1, scan throughput and seek latency are very important metrics that need to be optimized and evaluated. From Table II, we deduce that our new indexing scheme achieves substantial improvements in all categories compared to our previous one. We notice a 54% improvement in local (the client is in the same host with the HBase server responsible for the data) scan throughput and 100% improvement in remote scan performance (the client scan data from a remote HBase server). We also greatly reduce the latency of a seek operation due to the more compact representation of HBase key-values.

TABLE II. COMPARISON OF SCAN THROUGHPUT AND SEEK LATENCY

	RDF-3X	H_2RDF	H_2RDF+
Local Scan Throughput (million triples/sec)	17	0.73	1.13
Remote Scan Throughput (million triples/sec)	-	0.2	0.4
Seek latency cold cache (ms)	1	86	16
Seek latency hot cache (ms)	0.2	17	7

Compared to RDF-3X, scan/seek times are almost an order of magnitude larger. RDF-3X maintains extremely efficient clustered B^+trees that are placed in local disk storage. Our indexes suffer from retrieval overheads related to the distributed architectures of both HBase and HDFS. This performance overhead is alleviated by the capability of distributed, concurrent scanning inside MapReduce jobs.The impact of using distributed indexes will be made visible in the next section in the case of small, selective queries; the effect disappears when processing distributed, non selective joins.

TABLE III. PERFORMANCE COMPARISON OF H_2RDF+, H_2RDF AND HADOOPRDF FOR LUBM AND YAGO2 DATASETS

	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 IV. PERFORMANCE COMPARISON OF H_2RDF+ AND RDF-3X

	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	Import(min)	912	605	162	576	2075	1526	349	1398
YQ1(sec)	0.9	0.7	0.9	0.7	LQ1(sec)	0.6	0.4	0.6	0.3	0.8	0.4	0.9	0.4
YQ2(sec)	1.5	1	1.6	0.9	LQ4(sec)	2.1	0.8	2.1	0.7	2.3	0.8	2.2	0.8
YQ3(sec)	241	3037	138	1929	LQ2(sec)	373	2297	89	1277	706	Failed	119	2065
YQ4(sec)	123	2973	79	2068	LQ9(sec)	411	68	141	51	753	Failed	264	289

B. Direct Comparison

In order to provide a direct, fair comparison among the different systems, we first test the performance of H_2RDF+ versus the other distributed systems. We utilize four datasets, namely LUBM with 10k, 20k and 100k universities and Yago2, consisting of 1.3 billion, 2.7 billion, 13.8 billion and 120 million triples respectively. H_2RDF+, H_2RDF and HadoopRDF were executed using a cluster of 25 worker and 1 master nodes. In Table III we register the data import times and response times for the selected queries. 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 use as many worker VMs as the corresponding RDF-3X's server capacity allows. These results are reported in Table IV.

Data Import: This is the time needed for all systems to load the full dataset according to their storage scheme. 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, these jobs do not equally partition the reduce input data and thus overload some reducers while leaving others idle. H_2RDF+, despite its complex and more sophisticated indexing scheme, proves twice as fast as HadoopRDF. We also note that H_2RDF+ manages to import 3 additional indexes and keep more detailed statistics than H_2RDF at a mere 10-20% overhead. RDF-3X, being a centralized system, needs to parse *all* triples sequentially in order to create its indices. It also reads the input data several times. This iterative scan of input data results in an increasing import complexity: while the size of dataset doubles, the time needed to load the data triples. RDF-3X outperforms H_2RDF+ when running on a small server but fails to scale and proves 2–3 times slower when running with more resources.

Query Performance: LUBM provides a SPARQL query benchmark [3]. The compared systems do not support OWL reasoning so we only test queries that do not require reasoning or in some cases (e.g., query LQ9) we remove the hierarchy of the `rdf:type` predicate by querying for explicit types with no subclasses. Due to lack of space, we show results for five such queries (identified as LQ) that provide a good mixture of both simple and complex structures. The selected set covers all variations of the LUBM test queries; moreover they are able to highlight the different decisions and characteristics of H_2RDF+ and the compared systems. Yago2 does not provide benchmark queries; Relative to the LUBM queryset, we have created a set of representative test queries. In detail, there are two main categories of SPARQL queries tested: the ones that contain some selective pattern and have small number of results (LQ1, LQ3, LQ4, YQ1, YQ2) and the ones that contain no selective patterns and represent more complex join structures (LQ2, LQ9, YQ3, YQ4). A short description on the chosen queries is provided in the Appendix.

H_2RDF+ performs noticeably better in queries with large input: We exploit the orderings provided by our indexes via the distributed Merge and Sort-Merge join algorithms and achieve almost 7× performance gain compared to H_2RDF and 10× compared to HadoopRDF. We also outperform RDF-3X in most of the complex queries both when running on the small and the large server configuration. For example, for LQ2, RDF-3X requires almost 12GB of memory to execute the query for LUBM10k and proves 6× slower than H_2RDF+ in the small server setting. For LUBM20k (and large server setting) this increases to 14× slowdown compared to H_2RDF+. Our system achieves 3–6× smaller response times when moving to a larger cluster, while RDF-3X's speedup is mainly attributed to the bigger amount of memory (no swapping). In LQ9, RDF-3X manages to perform better, as it loads query data in memory. Yet, this approach does not scale; the system runs out of memory for LUBM20k on the small server.

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 lower scan throughput and the higher seek latency provided by our distributed HBase indexes. We also note that there is a small improvement compared to H_2RDF due to the more optimized indexing scheme. 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.

From these results, we deduce that H_2RDF+ processes all query types according to the goals set in its design: It manages to correctly identify selective vs. non-selective joins, performing either distributed or centralized joins, each join being performed in the most advantageous strategy. In high-selectivity queries, it is almost as efficient as RDF-3X, with a small difference (few tenths of a second) due to the fact that our index is shared across multiple cluster nodes. This small performance difference is alleviated by our system's ability to serve multiple concurrent queries (see Section IV-D). For more

data-intensive queries, it proves greatly superior to both central solutions and competitive Hadoop-based schemes due to both our join strategy and the ability to group multiple bindings.

LUBM full scale evaluation: Table III also contains H_2RDF+ and H_2RDF import and query execution times for the LUBM100k dataset that consists of 14 billion triples (2.5 TB), using a cluster of 1 master and 35 worker nodes. Our system achieves an import speed of 202 Ktriples/sec, a state-of-the-art performance according to [25]. Query response times follow the trend described in the previous experiments: For selective queries, centralized joins are selected, resulting in times that range between 0.8 and 2.4 sec. For non-selective queries with huge input sizes, such as LQ2 and LQ9, it achieves 3–4 times smaller response times compared to H_2RDF.

C. Join algorithm comparison

In this section, we compare the performance of our join algorithms over joins with different input sizes. 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. Fig. 3 shows the execution times required to join the full `ud:takesCourse` relation with the sampled one using different join algorithms. We range the sampled triples from 5 to 500 million.

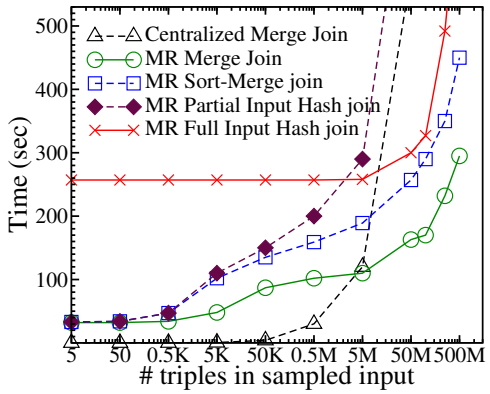


Fig. 3. Join algorithm scalability

We notice that for joins that contain one selective input triple pattern, the most efficient join strategy is the centralized Merge join algorithm. This is because MapReduce joins always incur an initialization overhead of almost 30 seconds. The performance of the centralized join deteriorates with the input size due to the fact that the algorithm does not exploit the parallel scanning capabilities of our distributed indexes.

Relative to MapReduce-based join algorithms, we consider the Merge, Sort-Merge, Partial Input Hash [23] and the Full Input Hash [23] join algorithms. We can clearly note that the Merge join algorithm has the best scalability performance due to the fact that it performs the join on sorted relations and minimizes the overhead of data movement. But this algorithm cannot be executed on intermediate, non-sorted relations. In this case, we can see that the Sort-Merge join proves to be the most scalable join algorithm. The difference between the Sort-Merge and the Partial Input Hash joins is the MapReduce partitioning method. The Sort-Merge join partitions the input data using a total order partitioner that takes advantage of the sorted indexes while the Partial Input join partitions using a Hash partitioner. This has impact on the reduce phase of the join: The Sort-Merge join performs a scalable merge join in the reduce phase while the Partial Input join executes a random HBase `get` on the indexes for each key. The second approach proves not scalable when the small input increases in size. Lastly, in the case that we have no sorted-indexed relation in the join, we need to fall back to the Full Input Hash join.

D. Concurrent execution of selective queries

For the case of centralized joins, we show that concurrent execution can result in very large query throughput. To achieve this, H_2RDF+ utilizes a zookeeper quorum that is responsible for the distribution of centralized joins to the cluster nodes. Each node has a maximum capacity of joins that can be simultaneously processed, set to 4 in our experiments. All tests are executed using the LUBM5k dataset. We use 10, 15, 20 and 25 worker nodes to see the impact of increasing the cluster size on the execution throughput. Results for queries LQ1, LQ3 and LQ4 are presented in Fig. 4. We present the average query throughput in queries per second. We run the same test twice to get the cold(CC) and warm cache(WC) throughput. We do not implement any special caching scheme but rely on HBase's caching. We notice that the warm cache execution results in 2 to 3 times higher throughput compared to the cold cache execution which means that our system can take advantage of caching. We observe an almost linear throughput increase to the number of worker nodes: For example LQ1 has a throughput of 65 q/sec (a 15.4 ms per query) in a 10-node cluster (40 executors). This is a speed-up of 40× as the individual execution of LQ1 takes 0.6 sec. LQ1 and LQ3 have almost the same performance due to their similar execution cost. H_2RDF+ needs 0.6 and 0.7 sec to answer LQ1 and LQ2 respectively. As for the smaller throughput of LQ4, this is due to its increased execution cost, as it needs approximately 2 sec to be answered. LQ4 exhibits the same scalability and warm cache properties discussed previously.

E. Query Scalability

In this section we evaluate the scalability properties of our distributed query processing. We use LQ9 because it is one of the most complex queries tested, requiring three distributed joins. We test query execution scalability using different dataset sizes and number of worker VMs. The scalability results for LQ9 are presented in Fig. 5. We test the performance of the MapReduce join execution using different dataset sizes using a 25-node cluster. The input and result size of LQ9 depends on the dataset size (directly affecting LQ9's execution time as well). Another parameter tested here is the region size effect on the MapReduce join execution. Large regions exhibit lower performance for small datasets because the number of tasks created fail to fully utilize the cluster resources. For larger datasets, all region sizes achieve good performance, as a result of having enough regions to fully utilize the cluster. For smaller region sizes (64MB or 32MB) the complexity is almost linear to the size of the input data.

Fig. 5 also shows the LQ9 query execution time as the number of nodes increases. All tests are executed using the

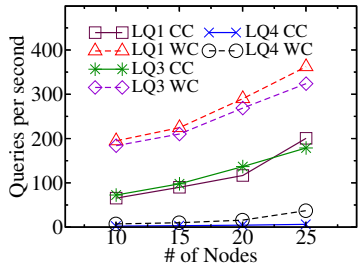
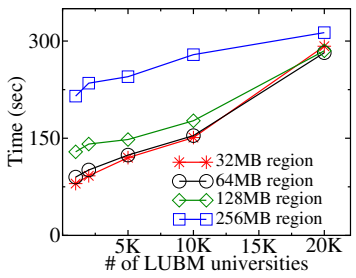


Fig. 4. Query throughput scalability



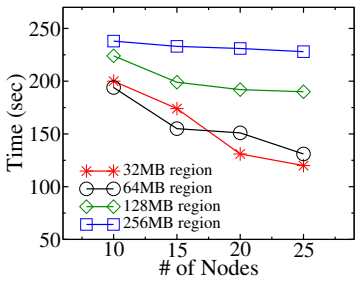


Fig. 5. Distributed query scalability for different number of universities and nodes for the LQ9

LUBM5k dataset. We vary the cluster size from 10 to 25 worker nodes and we vary the maximum region size from 32MB to 256MB. In the 32MB case, the join execution is highly scalable, gaining great speedup by adding more nodes. The deviations from linear speedup are mainly caused by the fact that the number of map tasks may or may not fit well to the cluster's capacity. As the region size grows, we note that adding more worker nodes does not significantly affect the speedup due to the fact that larger region sizes incur fewer tasks which cannot fully utilize cluster resources.

V. Conclusions

In this paper we presented H_2RDF+, a fully distributed RDF store capable of storing and querying arbitrarily large amounts of triples. The main contribution lies in our scalable distributed Merge and Sort-Merge join execution and our adaptive decisions about centralized and distributed join execution. We have also optimized both the compression and retrieval capabilities of our HBase indexes. H_2RDF+ greatly outperforms the compared centralized and distributed state-of-the-art RDF storage systems in non-selective multi-join queries, while being within a few tenths of a second to a state-of-the-art centralized engine in selective ones. H_2RDF+ is able to achieve great speedups and linear scaling in query processing and data loading tasks as well as high-throughput concurrent operations. These features allows H_2RDF+ to scale and handle non-selective queries in a dataset of size 2.5TB using a 35 small-sized worker node cluster.

Acknowledgment

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

References

[1] Google Snappy. https://code.google.com/p/snappy.
[2] HBase. http://hbase.apache.org.
[3] LUBM queries. http://swat.cse.lehigh.edu/projects/lubm.
[4] Resource Description Framework (RDF). http://www.w3.org/RDF/.
[5] Semantic Web. http://www.w3.org/standards/semanticweb/.
[6] Sesame. http://www.openrdf.org.
[7] SPARQL Query Language. http://www.w3.org/TR/rdf-sparql-query/.
[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] M. Atre, J. Srinivasan, and J. Hendler. BitMat: A Main-memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries. In *ISWC*, 2008.
[10] P. A. Bernstein and D.-M. W. Chiu. Using Semi-joins to Solve Relational Queries. *JACM*, 1981.
[11] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. SIGMOD '10.
[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] O. Erling and I. Mikhailov. Virtuoso: RDF Support in a Native RDBMS. In *Semantic Web Information Management*, pages 501–519. Springer, 2009.
[14] 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.
[15] S. Harris, N. Lamb, and N. Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. SSWS 2009.
[16] 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.
[17] 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.
[18] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 2011.
[19] M. Husain, J. McGlothlin, M. Masud, L. Khan, and B. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *TKDE*, 2011.
[20] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM - A Pragmatic Semantic Repository for OWL. In *WISE 2005*.
[21] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDBJ*, 2010.
[22] T. Neumann and G. Weikum. x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases. *VLDB*, 2010.
[23] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. *H_2RDF*: Adaptive Query Processing on RDF Data in the Cloud. In *WWW*, 2012.
[24] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^2Bench: A SPARQL Performance Benchmark. In *ICDE*, 2009.
[25] W3C. LargeTripleStores. http://www.w3.org/wiki/LargeTripleStores.
[26] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB*, 2008.
[27] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB*, 2013.

VI. Appendix

We provide the exact SPARQL queries used in the experimental section:

A. Yago2 Queries

YQ1: select ?x where { ?x y:livesIn y:Athens . }
YQ2: select ?y ?u where { ?y y:graduatedFrom ?u . y:Albert_Einstein y:graduatedFrom ?u . }
YQ3: select * where { ?p y:hasInternalWikipediaLinkTo ?p1 . ?p1 y:hasInternalWikipediaLinkTo ?p2 . ?p2 y:hasInternalWikipediaLinkTo ?p3 . }
YQ4: select * where { ?p y:hasInternalWikipediaLinkTo ?p1 . ?p1 y:hasExternalWikipediaLinkTo ?e1 . ?p y:hasExternalWikipediaLinkTo ?e . }

B. LUBM Queries

We utilize the queries provided in [3]. We only change LQ9 by removing OWL reasoning:
LQ9: select ?x ?z ?y where { ?x rdf:type ub:UndergraduateStudent . ?z rdf:type ub:FullProfessor . ?y rdf:type ub:Course . ?x ub:advisor ?z . ?x ub:takesCourse ?y . ?z ub:teacherOf ?y }