High Throughput Indexing for Large-scale Semantic Web Data

Long Cheng[1], Spyros Kotoulas[2], Tomas E Ward[3], Georgios Theodoropoulos[4]

[1] Technische Universität Dresden, Germany [2] IBM Research, Ireland
[3] National University of Ireland Maynooth, Ireland [4] Durham University, UK

long.cheng@tu-dresden.de, spyros.kotoulas@ie.ibm.com, tomas.ward@nuim.ie, theogeorgios@gmail.com

ABSTRACT

Distributed RDF data management systems become increasingly important with the growth of the Semantic Web. Currently, several such systems have been proposed, however, their indexing methods meet performance bottlenecks either on data loading or querying when processing large amounts of data. In this work, we propose an high throughout index to enable rapid analysis of large datasets. We adopt a hybrid structure to combine the loading speed of similar-size based methods with the execution speed of graph-based approaches, using dynamic data repartitioning over query workloads. We introduce the design and detailed implementation of our method. Experimental results show that the proposed index can indeed vastly improve loading speeds while remaining competitive in terms of performance. Therefore, the method could be considered as a good choice for RDF analysis in large-scale distributed scenarios.

1. INTRODUCTION

RDF stores are the backbone of the Semantic Web, allowing storage and retrieval of semi-structured information. Research and engineering on RDF stores is a very active area with many standalone systems such as Jena [15], Sesame [5], Hexastore [22], SW-Store [2] and RDF-3X [16] being introduced in the past years. However, as the size of RDF data increases, such single-machine approaches meet performance bottlenecks, in terms of both data loading and querying. Such bottlenecks are mainly due to (1) limited parallelism on symmetric multi-threaded systems, (2) limited system I/O, and (3) large volumes of intermediate query results producing memory pressure. Therefore, a system with efficient parallelization of data loading and querying based on distributed architectures becomes increasingly desirable.

Several approaches for distributed RDF data processing have been proposed [21, 19, 12, 14], along with clustered versions of more traditional approaches [11, 7, 4]. Depending on the data partitioning and placement patterns, distributed join processing can be divided into the following four categories. To better understand the basic idea of each approach in the following descriptions, we take an simple example, including four triples and two queries, which is shown in Figure 1 and Figure 2 respectively. We present the detailed implementation of each method over a two-node system and assume that terms with an odd number hash to the first node and constants with an even number hash to the second node (e.g. B1 hashes to node 1, B2 hashes to node 2).

SAC'15, April 3–7, 2015, Salamanca, Spain.

http://dx.doi.org/10.1145/2661829.2661888.

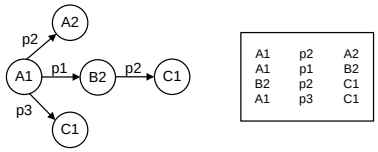


Figure 1: An RDF graph and the responsible triples.

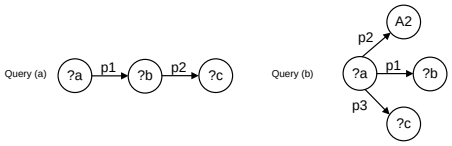


Figure 2: Two queries in the form of graph patterns.

Similar-size Partitioning. Systems based on similar-size partitioning place similar volumes of raw triples on each computation node without a global index. During query processing, nodes provide bindings for each triple pattern can be implemented in parallel, and the intermediate (or final) results can be then formulated by *parallel joins* [21]. Figure 3(a) shows the details of the partitioning that each node will hold two triples. Then during query execution, the solution mapping of each triple pattern will be located to a same node to implement local joins and consequently formulate the intermediate or final results. For example, for the Query(a) in Figure 2, the result of the first triple pattern <?a p1 ?b> at the first node <A1 B2> will be transferred to the second node, based on the hash value of the join key B2, to join with the <B2 C1> at the second node, and then output of the query result <A1 B2 C1>.

It can be seen that this schema has obvious performance advantages on data loading, as similar-size is very easy to achieve and each computing node can simply load its local data in parallel without inter-node communications. Re-

gardless, for any query including join operations, there will always be data movements in the specific implementations, which can consequently decrease the query performance, because network communication is always considered as the slowest operator in distributed data management systems deployed for large-scale analytics [18].

Hash-based Partitioning. Exploiting the fact that SPARQL queries often contain *star* graph patterns, triples under this scheme are commonly hash partitioned (by subject) across multiple machines and accessed in parallel at query time. As shown in Figure 3(b), the three triples with subject A1 are assigned to the first node while the other is assigned to the second node. Clearly, this kind of assignment will be more time cost than the above *similar-size* method, and there also exist same data movements when implementing the Query(a). However, when a query containing *star* pattern, for instance the Query(b) in the figure, then the included join operations will be totally computed locally, which can efficiently reduce the costly network communications and consequently improve the query performance.

Sharded/Partitioned Indexes. This approach is very closed to the centralized stores, triple indexes in the form of SPO, OPS etc. are distributed across all the computing nodes and stored locally as a B-Tree. Most of the existing parallel systems such as YARS2 [11], Clustered-TDB [17], Virtuoso-cluster [7] and 4store [10] belong to such a schema. Their operations are more similar to single-node RDF stores, normally offering lower loading speeds but can achieve persistence and more space-efficient indexing over a distributed system. Meanwhile, system I/O and join throughput of queries can be improved as well on that basis.

Graph-based Partitioning. Graph partitioning algorithms are used to partition RDF data in a manner that triples close to each other can be assigned to the same computation node. SPARQL queries generally take the form of graph pattern matching so that sub-graphs on each computation node can be matched independently and in parallel, as much as possible. Using such method, all the previous four triples will be placed on the same node based on a 2-hop graph (namely distance between two node is 2 maximum) as shown as Figure 3(c). Compared to the three approaches above, it can be seen that there will be no network communication for such a method during query executions, for both the queries in Figure 2. However, as graph partitioning is always complex, especially for large graph, the connections between each node will increase exponentially with increasing the graph, which could induce a very large time cost before loading the data.

In general, the techniques outlined above operate on a trade-off between loading complexity and query efficiency, with the earlier ones in the list offering superior loading performance at the cost of more complex/slower querying and the latter ones requiring significant computational effort for loading and/or partitioning. In fact, fast loading speed and query interactivity are important for exploration and analysis of RDF data at Web scale. For example, in a large-scale distributed scenario, large computational resources would be tapped in a short time, which requires very fast data loading of the target dataset(s). In turn, to shorten the data processing life-cycle for each query, exploration and analysis should be done in an interactive manner. To meet such a challenge, we are proposing a hybrid method for processing RDF using dynamic data re-partitioning to enable rapid analysis of large datasets.

Our approach combines both the similar-size and graph-based methods, and adopts a two-tier index architecture on each computation node for the implementation: (1) a lightweight primary index, to keep loading times low, and (2) a series of dynamic, multi-level secondary indexes, calculated during query execution, to decrease or remove inter-machine data movement for subsequent queries that contain the same graph patterns. This method is straightforward, yet not trivial, and have not been studied or evaluated. Consequently, the following three questions arising form the scheme are becoming to interested, in terms of performance:

- *hybrid*: using the approach, can we smoothly combine the loading speed of similar-size partitioning with the execution speed of graph-based partitioning, and achieve competitive performance with current solutions?
- *dynamic*: how will the dynamic construction of secondary indexes cost, is it worth to build such indexes so as to achieve runtime speedups in the presence of queries?
- *scalability*: will runtime of queries over the secondary indexes be scalable with increasing the number of computation nodes?

In this work, we introduce a hybrid and dynamic distributed RDF indexing approach, which specially targets fast loading data and computing queries on large RDF data, with a focus on analytical queries. We present the detailed design and implementation of the proposed method and conduct an experimental evaluation over a cluster with 16 nodes (192 cores). The results demonstrate that: (1) Our primary index results in very fast loading speeds, it takes only 7.4 minutes to load 1.1 billion triples, notably outperforming the single node system RDF-3X [16] and the cluster solution 4store [10]. (2) The secondary indexes significantly speed up query execution, bringing the performance of our implementation competitive to that of RDF-3X and 4store. Moreover, building secondary indexes is light-weighted and queries over the indexes are shown to be scalable.

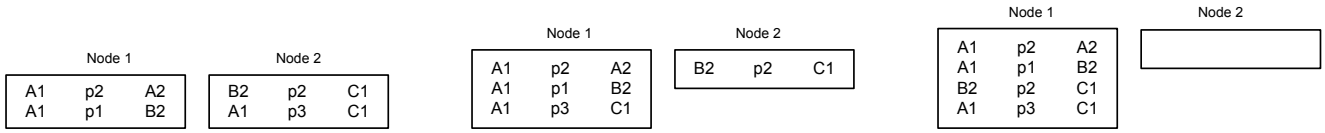
The rest of this paper is organized as follows: In the following Section, we present the design rationale and algorithms for our approach. In Section 3, we evaluate a prototype implementation and compare to RDF-3X and 4store. In Section 4, we reported on related work. Finally, in Section 5, we conclude the paper and point to directions for future work.

2. OUR APPROACH

We describe our approach in two parts, data loading and querying. The former includes *primary index building* while the latter focuses on *secondary index building*. We refer to the primary index as (l_1) and secondary indexes as 2nd-level (l_2), 3rd-level (l_3), etc.

2.1 Loading

As terms in RDF are represented by long strings, operating directly on them will result in (1) unnecessarily high space, memory and bandwidth consumption and (2) poor query performance, since computing on strings is computationally intensive. For converting the long strings to ids, we



(a) similar-size partitioning (b) hash-based partitioning (c) graph-based partitioning

Figure 3: Different kinds of RDF data partitioning over a two-node system.

take a similar dictionary encoding approach as the one described in [3]. Experimental results show that it has achieved higher throughput than any other methods in the literature [3]. Moreover, such method is more flexible for various semantic application scenarios, such as *transactional data processing* and *incremental updates*.

After encoding, we build the primary index l_1 for the encoded triples at each node. Similar to many triple stores, the index itself contains all the data. We use a modified *vertical partitioning* approach [1] to decompose the local data into multiple parts. Triples in [1] are placed into n two-column *vertical tables* (n is number of unique properties), which has been shown to be faster for querying than a single table. However, in [1], to efficiently locate data, all the *subjects* in each table are sorted, which is costly ($Nlog(N)$) in terms of data loading, especially when the tables are huge. In comparison, we only use linear-time operations for indexing, inserting each tuple in an unordered list in a corresponding *vertical table*. To support multiple access patterns, we build additional tables. By default, we build $P \rightarrow SO$, $PS \rightarrow O$ and $PO \rightarrow S$, corresponding to the most common access patterns.

For example, for the triples described in Figure 3(a), the first segment of Figure 4 shows the vertical tables of the primary index l_1, which is based on partitioning on the *predicate* and the *predicate-subject* of each encoded triple at each node (note that the triples are in the form of integers in this step, we use the *string* format in our examples just for readability). As each node builds their tables independently, there is no communication over the network for this step. Local indexing is very fast, so we could support additional indexes, e.g. to support more efficient joins on the predicate position, with minimal impact on performance.

As in all RDF stores, there is an element of redundancy in terms of data replication. Our index consumes more space than the vertical partitioning approach in [1], or a compressed index approach such as the one found in [16]. Nevertheless, our focus is on speed and horizontal scalability, which increases total available memory. In addition, based on the fast encoding method described above, the build process of the primary index is very lightweight: (1) triples are encoded and indexed completely in-memory and all accesses are *memory-aligned*, reducing CPU cost; (2) there is no global index as we only build an index for local data on each computation node, *reducing the need for communication*; (3) we avoid sorting, or any non-constant time operation, meaning that the *complexity of our approach is* $O(N)$, where N is the number of local statements; and (4) the encoding algorithm achieves good load balancing, which translates to good load balancing for the (local) indexing. The above factors contribute to very fast indexing, as we will show in our evaluation.

2.2 Querying

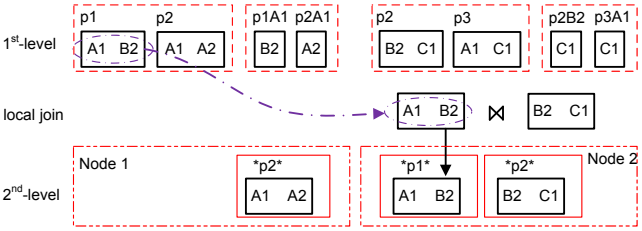


Figure 4: Query execution and the secondary index building.

Parallel Hash Joins. Once we have built the primary index, we can implement SPARQL queries through a sequence of lookups and joins. With the primary index l_1, we can easily look up the results for a statement pattern at each node. For example, for the two triple patterns in Figure 2, through looking up the vertical tables with the predicates $p1$ and $p2$, we can easily get the bindings for the variables ($?a$,$?b$) and ($?b$,$?c$) at each node:

	node 1	*node 2*
($?a$, $?b$)	(A1, B2)	/
($?b$, $?c$)	(A1, A2)	(B1, C1)

This lookup process can be implemented in parallel and independently for each node. Nevertheless, a *join* between any two sub-queries can not be executed independently at each node since we have no guarantee that join keys will be located on the same node. We adopt the parallel hash-join approach in our implementation. Namely, results of each subquery are redistributed among computation nodes by hashing the values of their join keys, so as to ensure that the appropriate results for the join are co-located [21]. Based on that, we redistribute the results of the two triple patterns by hashing bindings for the variable $?b$, and then implement the local joins for the received terms at each node. This process is shown in the first two segments of Figure 4.

Secondary Indexes. The local lookup for each triple pattern at each node is very fast, since we only need to locate the corresponding index table in l_1, and then retrieve all the elements. E.g. for the pattern <$?b$ $p2$ $?c$>, we can find the vertical table p_2 and return its results in constant time (since we use hashtables to index in the partitioned tables).

For join operations, as we have to redistribute all results for each triple pattern as well as the intermediate results, data transfers across nodes become costly, in terms of bandwidth and coordination overhead. To minimize data movement and improve query performance, we build secondary indexes (l_2 ... l_n), based on the redistribution of data during query execution. The build process of such indexes is presented in Algorithm 1. We have a queue of queries $\mathbb{Q}$.

Algorithm 1 Query Execution and Secondary Index Building

The primary index l_1 has been built, let $\mathbb{Q}$ be a query queue to be processed, l the secondary indexes initialized as $\emptyset$ at each node, r the intermediate results to be joined initialized as $\emptyset$.

```
   Main procedure:
 1: for each Q ∈ ℚ do
 2:     r=plan(Q)    //Plan query with root r
 3:     compute(r)
 4: end for

   Procedure compute(n):
 5: r_i = l.lookup(n)
 6: if r_i ≠ null then
 7:     return r_i    // If an index already has the result
 8: else
 9:     for each child c in n parallel do
10:         if c is a triple pattern then
11:             lr_i=l_1.lookup(n)
12:             r_c=redistribute(lr_i)
13:         else
14:             r_c=compute(c)
15:         end if
16:         r.add(r_c)
17:         if isIndexable(r_c) then
18:             l.index(c,r_c)
19:         end if
20:     end for
21:     return join(r)
22: end if
```

For each query Q, we assume a planning method (which is beyond the scope of this paper) that results in an execution plan represented as a tree with root r. We assume that queries in the queue are processed sequentially and each node keeps a set of indexes of various levels $l_{1..n}$. All nodes start with index l_1 built and all other indexes empty.

We evaluate the expressions in the tree bottom-up, in parallel (lines 9 and 14), redistributing results as required (line 12). The function `isIndexable()` determines whether nodes should retain the (indexed) data from remote nodes. The construct `parallel do` implies synchronization at `end for`. Results from existing indexes are re-used when possible (lines 6 and 7). Once the results of all children of a node become available, a join is executed. Note that this process implies a high degree of parallelism since individual joins are executed in parallel and multiple join expressions are calculated in parallel, when possible. From example, as demonstrated in the third segment of Figure 4, a set of new tables is built on l_2: for $*p1*$ and $*p2*$, when we first implement the query.

It can be seen that the building process is straightforward that the index is constructed just by a simple *copy* of the redistributed data, which is introduced by a *join* of a query. Namely, it is a byproduct of query execution. Regardless, this index is efficient on improving query performance in a analysis environment, because it can be reused by other queries that contain patterns in common. We are using the term indexing instead of caching, because the data is re-partitioned on demand and is fully indexed in a sharded manner, as opposed to storing intermediate results and re-using them, such as the *cache* used in centralised RDF stores [20]. This means that indexes can be re-used for any query containing them and the consequent cost is that we need to re-compute the joins locally.

Index Levels. According to Algorithm 1, the k-th level index l_k is built based on the redistribution of the data stored in the level $k-1$. In the meantime, if a query is indexed by the index l_k, the the execution of joins in this query will be cost-free in terms of network communication. This means that, there will be only local joins for the query then.

In the process of building the k-th level index l_k, if we run all possible queries, what will the data on each node look like? In fact, according to the terminology regarding *graph partitioning* used in [12], the 2nd-level index in our method on each node will construct a 2-hop subgraph, the 3rd-level one will be a 3-hop subgraph, and l_k will be a k-hop subgraph. For example, the two triple <A1 p1 B2> and <B2 p2 C1> at the second node of Figure 4 construct an instance of the 2-hop subgraph. This means that our method essentially does dynamic graph-based partitioning starting from an initial equal-size partitioning, based on the query load. Therefore, our system can combine the advantages of fast data loading and efficient querying. We will show that this design is indeed efficient in our evaluation presented in Section 3. In addition, the theoretical results from [12] can be applied for our approach as well.

Secondary indexes l_k can reduce/remove the network communication for a query. As k increases, the transferred data between nodes decreases, resulting in improved performance. However, the space for the entire index l also increases, constituting a trade-off between space and performance. It is possible to use the method *discriminative and frequent predicate path* presented in [23] to reduce the size, regardless, this is beyond the scope of this work.

3. EVALUATION

In this section, we present an experimental evaluation of our approach and compare its performance with a top-performing RDF store running on a single node as well as a cluster RDF store.

Platform. We use 16 IBM servers with each containing two 6-core Intel Xeon X5679 processors clocked at 2.93 GHz, 128GB of RAM and a single 1TB SATA hard-drive, connected using Gigabit Ethernet. We use Linux kernel version 2.6.32-220, X10 version 2.3 compiled to C++ and gcc version 4.4.6.

Setup. We implemented our approach with the X10 parallel programming language [6][1]. We have taken RDF-3X [16] and 4store [10] for the performance references of our implementation. The former represents the state-of-the-art in terms of single machine stores, which is widely used for comparison in recent solutions [12, 25]. The latter is a clustered RDF store, which is designed to operate mainly in memory[2]. To focus on analyzing the core performance of query execution, we only counter the number of results but not output them. We do not compare with MapReduce-based approaches since, due to platform overhead, they do not execute interactive queries in reasonable time. For example,

[1]Our method can be implemented in any programming languages, such as MPI or C++.

[2]Refer to `http://4store.org/trac/wiki/Tuning`

Table 1: Time to load 1.1 billion triples

System	Loading time (s)	Throughput triples /sec	Throughput per node
RDF-3X	**23296**	47.2K	47.2K
4store	**7078**	155.4K	9.7K
Our method	Read from disk: 103 Triple encoding: 254 Building l_1: (P, PO, PS) 86 Total: **443**	2483.1K	155.2K

SHARD [19], has runtimes for LUBM in the hundreds of seconds.

Benchmark. We load LUBM(8000), containing about 1.1 billion triples (about 190GB) and run all 14 queries on this data. As our system does not support RDF inference, we use a modified query set to get results for most queries[3]. For example, since the basic graph pattern <?x type Student> returns no results in Query 10, we use <?x type GraduateStudent> instead.

We are focusing on an indexing method as opposed to a full clustered RDF store in this work, therefore, we have chosen a relative simple benchmark in our test - LUBM [8], which includes BGPs with varying selectivity and complexity, and also have been adpoted in recently distributed stores [12, 25, 9]. To conduct a fair performance comparison, we load and query data in memory, so as to reduce the effect of I/O. Therefore, we set the index locations of RDF-3X and 4store to a `tmpfs` file system resident in memory at each node, so that queries can be fully implemented over distributed memory. For data loading, because our `tmpfs` file system at each node can not hold all 1.1 billion triples, we load data from hard disk to memory for the two stores. Although our system can operate completely in the distributed memory, in the interest of a fair comparison, we read data from disks as well during the data loading process.

3.1 Loading

We load 1.1 billion triples and build three primary indexes (on P, PO and PS). For RDF-3X and 4store, we report the time to bulk load data from disk into the memory partition(s). For both systems, we are using the default indexes.

As shown in Table 1, our system takes 103 seconds to read the data into memory, 254 seconds to encode triples and 86 seconds to build the primary index l_1, for an average throughput of 429MB or 2.48M triples per second. In comparison, 4store takes 7078 seconds[4], for an average throughput of 155K triples per second. The reason is that our loading process is fully parallel and our indexes are very lightweight, while 4store needs to do global sorts and uses a master node for coordination.

We also see that RDF-3X takes about 6.5 hours, for an average throughput of 47K triples per second, performing much worse than the other two systems (presumably because we are running on one node and because of the heavier indexing scheme of RDF-3X). From the results reported in [12], the graph-based partitioning method (used for parallel solutions) is even slower than RDF-3X, which highlight the advantage of our approach again, in terms of loading speed.

[3]The rewritten queries can be found at the same link of our code.

[4]Though 4store is a quad-store and has to index graphs IDs, there is only one graph in the dataset and the overhead is very small.

3.2 Querying

Runtime. To test how fast can we achieve on querying, we execute all LUBM queries using l_1 and l_2, since the number of joins in most queries is small. Although our system does not use a cache as such, one could consider executions with secondary indexes as warm runs and l_1 as a cold run (we explain further regarding the costs and benefits of additional index levels later in this section).

Table 2 shows the execution time for each query. Both RDF-3X and 4store are very fast for most queries, staying under 1ms, since many queries in LUBM are very simple. There is only a marginal difference between cold and warm runs, since we are operating in memory. In our system, the execution over l_2 is generally much faster than over l_1, which shows that query performance can be vastly improved by building a secondary index. The lowest speedup is achieved on Q2, Q9, Q6 and Q14, the reasons being that (1) Q2 and Q9 are complex and the intermediate results still need redistribution over the l_2 index; and (2) Q6 and Q14 contain only a single triple pattern, thus l_2 is not built.

Comparing the warm run of RDF-3X and our implementation with the 2nd-level index: (1) our approach is slower than RDF-3X for simple and selective queries such as Q1 and Q3. RDF-3X uses some hundreds of μs to finish the operations of lookup and joins for candidate results while our system (and 4store) has to do synchronization over a distributed architecture, which has an overhead of about 10 *ms*; (2) our system is much faster at *complex queries*, for example Q2 and Q9, as we can implement joins in parallel; and queries having *low selectivity*, for example Q6 and Q14, since it has higher aggregate I/O; or possibly both reasons, such as Q13.

Meanwhile, compared to 4store, we are slower for some queries, such as for the Q1, Q5, Q6, Q10, Q11 and Q13. Regardless, the difference of the time cost is very small, only in the order of *ms*. The possible reason could be the overhead of our join operations but not our indexing approach, because we only adopt *hash join* as local joins in our implementation and we have to build hash tables firstly which are then probes. We are also slower on Q4, Q7, Q8 and Q12, in the order of *100 ms*, which could be because 4store optimizes the coordination between each node, while our system currently involves all nodes in each query. However, the much faster loading time, in combination with the fact that our approach always stay in the interactive range, makes our approach better suited for some applications.

For the more complex queries Q2 and Q9, our approach is obviously much faster, in the order of *sec*. Moreover, we can further improve the performance of our system by employing higher level indexes. On the other hand, our method is also faster than 4store for the simple queries Q3 and Q14. The reason could be that we can quickly locate required indexes and then organize scans for large number of tuples (for Q14) or the used *local hash join* demonstrates its advantages on small-large table joins (for Q3).

Indexes. We examine the time cost to build the secondary indexes, and examine query performance on executing Q2 and Q9, which are the most complex queries containing more triple pattern than others. Figure 5 shows that building a

Table 2: Execution times for the LUBM queries over RDF-3X and 4store with cold and warm runs, as well as our system with the primary index l_1 and second-level index l_2 (ms)

Q.	RDF-3X		4store		Our system		#
	cold	warm	cold	warm	l_1	l_2	Results
1	0.19	0.17	9	8	500	14	4
2	11303	11217	4635	4510	8244	3917	2528
3	0.26	0.25	24	22	1635	20	6
4	0.34	0.28	0.45	0.32	10597	445	10
5	0.22	0.18	4.08	3.57	1012	13	146
6	409	382	6.49	5.71	12	12	20 mil.
7	0.64	0.54	0.19	0.15	8129	731	0
8	1.73	1.55	0.69	0.64	5145	564	1874
9	10253	9803	18148	17972	9533	4173	0
10	0.21	0.17	5.76	4.79	986	15	4
11	0.21	0.17	1.24	1.20	505	13	0
12	125	124	0.24	0.20	1285	384	125
13	202	199	18.49	16.01	1141	18	19905
14	1147	1055	21.19	20.45	16	16	63 mil.

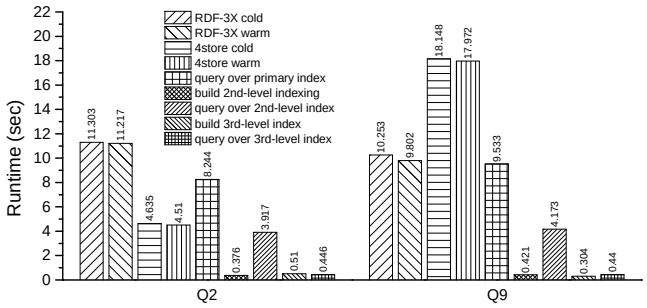


Figure 5: Runtime for RDF-3X and 4store, and detailed runtime of each implementation for our approach (over Q2 and Q9 using 192 cores).

Table 3: Runtime by varying the number of cores over 2nd-level index

# nodes	12	24	48	96	192
Q2	20.804	15.613	13.027	6.827	3.917
Q9	11.453	9.516	7.908	5.272	4.173

high-level index takes only hundreds of *ms*, which is extremely small compared to the query execution time. This operation is very fast, since it only involves indexing using in-memory hashtables. We can also see that, the higher the level of index is, the lower the execution time. For example, with l_3, Q2 and Q9 can be executed in 0.45 seconds, which is orders of magnitude faster than with l_2, RDF-3X and 4store. The reason is that, for l_3, there is no data movement between nodes for joins and we only need to perform local joins.

Scalability. We also test the scalability of our implementation by varying the number of processing cores. We run Q2 and Q9 over the second-level index and double the number of cores from 12 (a single node) till 192. The results are presented in Table 3. It can be seen that the execution time of both queries decreases with increasing the number of cores. Nevertheless, both queries reach a plateau at around 4 seconds. The reason for this is that overhead starts dominating the runtime. With 192 cores, for each core, there will be approximately 191 (one from each other node) messages, with the associated coordination overhead, for a total of 532K and 375K tuples transferred for Q2 and Q9 respectively. As future work, we will work on methods to reduce the distribution for small indexes, so as to avoid this messaging and coordination overhead.

4. RELATED WORK

RDF processing systems geared towards batch processing [21, 14] are based on architectures developed for a similar-size data partitioning model. In this respect, these systems are similar to the one proposed here in terms of fast data loading and minimal or no pre-processing. However, they execute queries directly over the *raw* data without any encoding process or additional index, resulting in a heavy network communication costs for complex queries and significant startup overhead. For example, while [14] can process massive datasets with zero loading time, its minimum runtime is in minutes, not seconds.

Systems such as SHARD [19] and the one in [13] generally adopt hash-based partitioning techniques. This leads to slower loading of RDF data, e.g. 0.5 hour to load 270 million triples is reported in [12]. These systems are similar to our system using the 2nd-level index. Therefore, they can avoid communication for simple queries containing star graph patterns. For complex queries with higher-level operations, our system is much faster, because large amounts of data in these systems still needs to be redistributed across the network to perform joins.

Clustered RDF stores such as Virtuoso Cluster [7], YARS2 [11] and 4store [10] distribute indexes over nodes in a cluster to improve I/O and join throughput. They are more similar in operation to single-node RDF stores than to our approach, offering lower loading speeds but also persistence and more space-efficient indexing. As shown in our tests, we are much faster than 4store in data loading and also outperform it for complex queries.

Systems using graph-based partitioning such as the ones in [12, 24, 25], are similar to the ones using high-level indexes proposed here, which impacts positively on query performance. However, graph partitioning and triple placement in these systems happens at indexing time, hampering loading throughput. For example, the system described in [12] takes 4 hours to assign 270 million triples according to a 2-hop construction. Although [25] stores data as a graph, time spent on graph partitioning will still increase exponentially with increasing either the size of a graph or the parameter *hop*, because the connections between vertexes becomes more complex. In contrast, our system has no such costly operations, but organizes the sub-graph dynamically. Moreover, our incremental indexing process has proven to be very lightweight, requiring only hundreds of *ms*, in addition to query execution time.

5. CONCLUSION

In this work, based on the analysis of current indexing approaches, we present an efficient hybrid structure designed for fast loading and querying large-scale RDF data over distributed systems. We implement our approach over a commodity cluster and the experimental results demonstrate that our approach is extremely fast at loading data while still keeping query response time within an interactive range. Future work lies in further extensions to our design through the application of methods for *index size reduction* (or index management) and *sort-based* local joins, to develop a highly scalable distributed analysis system for extreme-scale RDF data.

Acknowledgments. Long Cheng was supported by the DFG in grant KR 4381/1-1. The computations were performed on the High-performance Systems Research Cluster at IBM Research Ireland.

6. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[2] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: A vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 18(2):385–406, Apr. 2009.

[3] Anonymous. In *To avoid compromising the double-blind review process, we have removed a citation to an accepted by not yet published article here.*

[4] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. OWLIM: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011.

[5] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC*, pages 54–68. 2002.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices*, 40(10):519–538, 2005.

[7] O. Erling and I. Mikhailov. Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management*, pages 501–519. Springer, 2010.

[8] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[9] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. *SIGMOD*, pages 289–300, 2014.

[10] S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In *SSWS*, pages 94–109, 2009.

[11] A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In *The Semantic Web*, pages 211–224. 2007.

[12] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.

[13] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. Heuristics-based query processing for large RDF graphs using cloud computing. *TKDE*, 23(9):1312–1327, 2011.

[14] S. Kotoulas, J. Urbani, P. Boncz, and P. Mika. Robust runtime optimization and skew-resistant execution of analytical SPARQL queries on PIG. In *ISWC*, pages 247–262. 2012.

[15] B. McBride. Jena: Implementing the rdf model and syntax specification. In *SemWeb*, 2001.

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

[17] A. Owens, A. Seaborne, N. Gibbins, et al. Clustered TDB: a clustered triple store for Jena. 2008.

[18] O. Polychroniou, R. Sen, and K. A. Ross. Track join: distributed joins with minimal network traffic. In *SIGMOD*, pages 1483–1494, 2014.

[19] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework: The SHARD triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, page 4, 2010.

[20] J. Umbrich, M. Karnstedt, A. Hogan, and J. X. Parreira. Hybrid SPARQL queries: fresh vs. fast results. In *ISWC*, pages 608–624. 2012.

[21] J. Weaver and G. T. Williams. Scalable rdf query processing on clusters and supercomputers. In *SSWS*, page 68, 2009.

[22] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, Aug. 2008.

[23] X. Yan, P. S. Yu, and J. Han. Graph indexing based on discriminative frequent structure analysis. *TODS*, 30(4):960–993, 2005.

[24] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *SIGMOD*, pages 517–528, 2012.

[25] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4):265–276, 2013.