Scale-Out Processing of Large RDF Datasets

Long Cheng and Spyros Kotoulas

Abstract—Distributed RDF data management systems become increasingly important with the growth of the Semantic Web. Regardless, current methods meet performance bottlenecks either on data loading or querying when processing large amounts of data. In this work, we propose efficient methods for processing RDF using dynamic data re-partitioning to enable rapid analysis of large datasets. Our approach adopts a two-tier index architecture on each computation node: (1) a lightweight primary index, to keep loading times low, and (2) a series of dynamic, multi-level secondary indexes, calculated as a by-product of query execution, to decrease or remove inter-machine data movement for subsequent queries that contain the same graph patterns. In addition, we propose methods to replace some secondary indexes with distributed filters, so as to decrease memory consumption. Experimental results on a commodity cluster with 16 nodes show that the method presents good scale-out characteristics and can indeed vastly improve loading speeds while remaining competitive in terms of performance. Specifically, our approach can load a dataset of 1.1 billion triples at a rate of 2.48 million triples per second and provide competitive performance to RDF-3X and 4store for expensive queries.

Index Terms—RDF data; dictionary encoding; hybrid index; index building; distributed filter; big data

1 INTRODUCTION

Governments and enterprises are increasingly seeing the benefits of RDF regarding interoperability and flexibility for data representation and processing (e.g., [1]). In addition, the exploding availability of RDF datasets from multiple domains such as Linked Data [2] is challenging today's processing techniques.

RDF. The Resource Description Framework (RDF) [3] is a schema-less, graph-based data representation. It entails subject-predicate-object (SPO) expressions describing resources and their relationships. These expressions are known as RDF triples. For instance, the statement from DBpedia (<dbpedia:IBM>, <dbpedia-owl:foundation-Place>, <dbpedia:New-York>) conveys the information that the corporation IBM was founded in New York. The Semantic Web already contains tens of billions of such statements and this number is growing rapidly.

In RDF, a set of triples can be represented as a directed labeled graph termed as an RDF graph. For example, Figure 1 illustrates an RDF graph with four triples. In such an RDF graph, each subject and object of a triple is represented as a vertex, and the predicate is described as a labeled directed edge from the responsible subject to the object. Note that, all the vertexes in a graph are unique regardless of the number of appearances for a subject, predicate or object in the underlying triples. Namely, the same subject or object from different RDF triples is represented by the same vertex. It should be noted that the RDF representation allows edges between predicates technically, but this is beyond the scope of this paper, and the experiments presented herein.

SPARQL. SPARQL is the standard RDF query language that facilitates the extraction of information from stored RDF data. The core component of SPARQL queries is a conjunctive set of *triple patterns*, termed a Basic Graph Pattern (BGP). Similar to an RDF triple, a basic triple pattern is also in the form of subject-predicate-object, the difference is that any component of the pattern could be a variable. A basic triple pattern could match a subset of the underlying RDF data, where the terms in the triple pattern respond to the ones in the RDF data [4]. Consequently, a solution mapping is defined as the mapping from the variables to the respective RDF terms in the data.

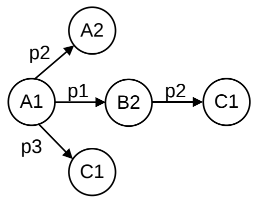
In addition, a SPARQL query can also be thought of as a graph called a *query graph pattern*. For example, Figure 2 shows two queries in the form of graph patterns, which contain two and three basic triple patterns respectively. In this scenario, the implementation of a SPARQL query can be also considered as a subgraph matching process.

Challenge. 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 [5], Sesame [6], Hexastore [7], SW-Store [8] and RDF-3X [9] 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 [10], [11], [12], [13], along with clustered versions of more traditional approaches [14], [15], [16]. These systems have already achieved significant performance improvements on either data querying or loading, however, not both aspects, as we will explain in the following section.

In fact, fast loading speed and query interactivity are

- *L. Cheng is with the Faculty of Computer Science, TU Dresden, Germany. E-mail: long.cheng@tu-dresden.de*
- *S. Kotoulas is with IBM Research, Dublin, Ireland. E-mail: spyros.kotoulas@ie.ibm.com*



A1	p2	A2
A1	p1	B2
B2	p2	C1
A1	p3	C1

Fig. 1. An RDF graph and the responsible triples.

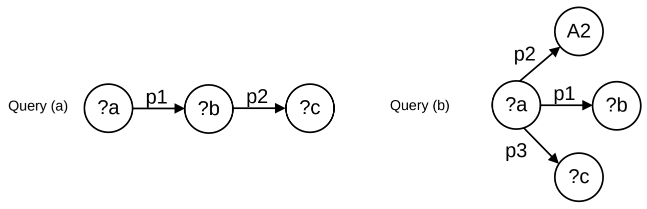


Fig. 2. Two queries in the form of graph patterns.

important for exploration and analysis of RDF data at Web scale. For example, in a large-scale distributed scenario exploiting Cloud infrastructure, vast computational resources could be tapped for a short amount of time. This would require very fast data loading of the target dataset(s) so as to decrease cost. In addition, to shorten the data processing life-cycle for each query, exploration and analysis should be done in an interactive manner.

Our approach. To meet these challenges, we are proposing a hybrid method for processing RDF using dynamic data re-partitioning to enable rapid analysis of large datasets. Our approach combines the two most commonly used methods applied in current systems: similar-size partitioning and graph-based partitioning, where the former approach can fast load data while the latter is efficient on query processing. We adopt a two-tier index architecture on each computation node for our 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. In the meantime, we also investigate approaches to reduce the sizes of multi-level secondary indexes, as they grow in size.

This method is straightforward, in terms of system complexity, yet not trivial, and complementary to caching techniques. Whereas our approach aims at re-using the re-organisation of data that happens during query execution, caching techniques store results of queries or parts of queries.

We focus on the following research questions:

- *Hybrid index*: Can we combine the loading speed of similar-size partitioning with the execution speed of graph-based partitioning, and achieve competitive performance with current solutions?
- *Dynamic index construction*: What does the dynamic construction of secondary indexes cost, in terms of computation and memory? When is it worth building such indexes, in terms of cost and query execution speedup?
- *Scalability*: Does the runtime of queries over the secondary indexes scale well with increasing the number of computation nodes (scale-out properties)?

We aspire that the answers to these questions will inform readers developing systems that balance loading speed to query execution speed for processing large RDF datasets.

Contribution. This manuscript is an extension of our previous work [17]. Specifically, we have introduced significant extension for our previous approach - distributed filters, as an optional replacement for secondary indexes, so as to reduce memory consumption. We believe that this extension constitutes an important addition, in terms of improving memory consumption, which could be a key criticism to our approach otherwise. Moreover, we also present more detailed experimental results and analysis in this work.

We summarize the contribution of this paper as follows: (a) We present a dynamic distributed RDF indexing approach that can both load data and compute queries quickly on large RDF data, with a focus on analytical queries. (b) We implement a fully parallel version of our approach and evaluate its performance on a cluster using the LUBM benchmark [18]. (c) Experimental results show that constructing only a primary index results in very fast loading speeds: It takes only 7.4 minutes to load 1.1 billion triples on 16 nodes, for a throughput of 2.48 million triples per second, notably outperforming other systems (e.g., 4store [19]). (d) In the meantime, our secondary indexes show their scalability and can significantly speed up computation, bringing the performance of our approach close to that of RDF-3X and 4store: It takes about 9 seconds, 4 seconds and 0.4 seconds to execute the two most complex LUBM queries over our primary, 2nd level and 3rd level indexes respectively. This performance is better than a state-of-the-art RDF store (RDF-3X) operating on a single machine in main memory for both queries. It is also faster than a clustered RDF store operating in memory (4store) when using the 2nd level and 3rd level indexes. For the other queries and indexes, our approach still stays within an interactive response time, although most are slower than the other systems. (e) In addition, our tests also show that, by using an additional optimisation, namely filters, we can achieve 1.14 - 3.45x execution speedup, with minimal storage overhead (up to 1.2% of index size).

The rest of this paper is organized as follows: In the following Section, we describe current approaches and discuss about their possible performance issues. In Section 3, we present the design rationale and algorithms for our approach. In Section 4, we evaluate a prototype implementation and compare its performance to RDF-3X and 4store. In Section 5, we reported on related work. Finally, in Section 6, we conclude the paper and point to directions for future work.

2 Current Approaches

Depending on the data partitioning and placement patterns, current distributed RDF systems can be divided into the following four categories. To better understand the basic idea of each approach, in the following descriptions, we take a simple example consisting of four triples and two queries as shown in Figure 1 and Figure 2 respectively. We present

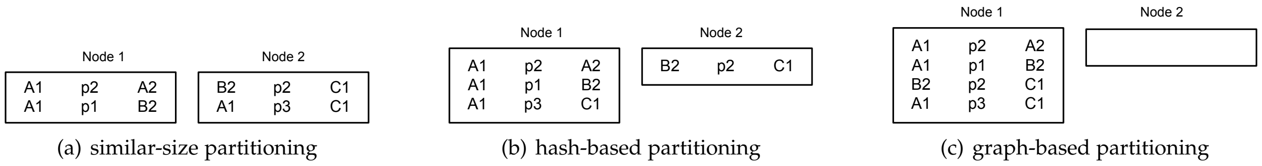


Fig. 3. Different kinds of RDF data partitioning over a two-node system.

the detailed implementation of each method over a two-node system and assume that terms with an odd number are hashed to the first node and constants with an even number are hashed to the second node (e.g., B1 hashes to node 1, B2 hashes to node 2).

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* [10], [13]. 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 scheme 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 communication. For example, in [13], no discrete loading phase is necessary, as the approach relies on the distribution of data on a distributed filesystem. Regardless, 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 [20].

Hash-based Partitioning. This method exploits 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 costly than *similar-size* partitioning, and there still exists some data movement when implementing the Query(a). However, for a query containing a *star* pattern, for instance the Query(b) in the figure, the included join operations will be computed locally, which can vastly reduce costly network communication and consequently improve query performance. As an example, in [12], it is reported that hash-partitioning takes 30 minutes for a 270 million-triple dataset (8 times faster than the graph partitioning presented in the same work).

Sharded/Partitioned Indexes. This approach is very similar to how centralized stores operate. 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 [14], Clustered-TDB [21], Virtuoso-cluster [15] and 4store [19] use such a scheme. Their operations are more similar to single-node RDF stores, usually offering lower loading speeds but achieving persistence and space-efficient indexing. Meanwhile, system I/O and join throughput of queries can be improved as well on that basis. In our evaluation, a partitioned index-based approach is around 5 times slower per node, compared to a centralized store, although much faster across all nodes.

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 methods, 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 execution, for both the queries in Figure 2. However, as graph partitioning is always complex, especially for large graphs, loading time can be very significant, and the partitioning approach can result in excessive memory consumption (since triples will usually belong to multiple partitions). In one system [12], graph partitioning and loading on a 20-node cluster was slower than loading data without partitioning on a single machine.

Discussion. 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. More importantly, the cost of advanced partitioning methods is amortized over the anticipated query workloads, compared with partitioning the entire dataset. As shown in [22], only a small fraction of the whole underlying data is actually accessed by typical real query workloads. For instance, a real workload consisting of thousands of queries executed over DBpedia [23] touches only 0.003% of the entire dataset [24], although this work focuses on analytical workloads, rather that queries posed on Web endpoints. In any case, a dynamic and adaptive partitioning method based on query workloads seems more attractive. In Section 5, we compare against specific approaches.

3 Our Approach

We are proposing a set of parallel techniques that combine the loading speed of similar-size partitioning with the execution speed of graph-based partitioning. The main elements of our approach are as follows:

- We use fixed-length integer encoding for RDF terms and constant-time operations for indexing (i.e., indexes are based on hash-tables), to increase access speed.
- During indexing, we do not use network communication, to increase loading speed.
- We maintain a local lightweight primary index supporting very fast retrieval, so avoid costly scans.
- We use secondary indexes supporting non-trivial access patterns that are built dynamically, as a byproduct of query execution, to amortize costs for common access patterns.
- We optionally reduce secondary indexes into filters, to reclaim memory.

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*. For conciseness, we refer to the primary index as (l_1) and secondary indexes as 2nd-level (l_2), 3rd-level (l_3), etc. in the following.

3.1 Loading

3.1.1 Statement Encoding

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 (i.e., integers), we take a similar approach as we have described in [25], [26].

Consider the four RDF statements described in Figure 1. We utilise a distributed dictionary encoding method for the input data, transforming RDF terms into integers and representing statements using this encoding. Same as the similar-size partitioning methods, the data is first divided into a number of equal-size *chunks* and then assigned as input for processing on separate computation nodes (i.e., Figure 1(a)). Then, the overall implementation strategy for each node can be divided into three steps as follows.

Step 1. Each statement is firstly parsed and split into individual terms, namely, subject, predicate and object. In this process, the duplicated terms are locally eliminated by a filter operation, and the extracted set of *unique* terms is divided into individual groups according to their hash values[1]. The number of groups is set to be the same as the number of nodes, and terms with the same hash are placed in the same group. After that, according to the hash values, each group will be sent to the corresponding remote node for the following dictionary encoding. In this process, we use the hash value modulo the rank of the node to assign keys to nodes. For example, the terms in the first group (namely A1,p1) will be sent to the first node itself and others are send to the second node. The detailed operations of this step on the first node is shown as below.

parsing	[A1,p2,A2,A1,p1,B2] ⇒
filter	(A1,p1,p2,A2,B2) ⇒
hash-groups	{A1,p1} + {p2,A2,B2}

Step 2. The term encoding process can commence once the grouped unique terms have been transferred to the appropriate remote nodes. The detailed encoding implementations at each node is very similar to a sequential approach. Namely, each received term access the local dictionary sequentially to get its numerical ID. In this process, if the mapping of a term already exists, then its ID is retrieved directly. Otherwise, a new ID will be created for the term, and the new mapping will be added into the local dictionary. In both cases, the ID of the encoded term will be kept in memory in groups. Once all the received terms have been encoded, all the ids will be sent back to the requester(s). To guarantee that there is no clash between term IDs assigned at different nodes, the value of a new ID is determined by the summation of the largest ID in the dictionary and the number of nodes n and the initial ID for each node is set as its rank. Moreover, each ID is formatted as an unsigned 64-bit integer in order to remove limitations regarding maximum dictionary size[2]. In this case, the first node will receive the IDs as following.

send	{A1,p1} + {p2,A2,B2}
receive	{1,3} + {2,4,6}

Step 3. The statements at each node can be encoded after all the IDs of the pushed terms have been pushed back. Since the terms and their respective IDs are held in order inside arrays in our approach, we can easily insert these mappings into a local dictionary to encode the parsed triples kept in the first step. Each of the three steps is implemented in parallel at each node, and the whole encoding process terminates when all individual nodes terminate. Namely, we will get the encoded triples shown as below at the first node.

parsed	[A1,p2,A2,A1,p1,B2] ⇒
encoded	<1 2 4>, <1 3 6>

During the entire encoding process, a *filter* structure (we used a `HashSet` in our experiments) is employed to process the terms and to extract the unique terms that need to be transferred to the remote node. This is done for all terms irrespective of their popularity. Using the filter guarantees that any given term can possibly move to a remote place just once per node, which will be very effective when handling the data skew for RDF terms. For example, although the term *type* appears highly frequent, each node will transfer only one such term maximally to remote node using our approach, which can drastically reduce inter-machine communication and local computation. Moreover, we use an advanced two-sided communication pattern that we only send terms and retrieve their IDs (instead of retrieving <term, id>), vastly reducing network cost when transferring a large number of strings (note again that RDF terms are always

1. The hash value of each term is assigned by the common used RS hash algorithm [27].

2. Note that it is possible to use arbitrary- or variable-length IDs in order to further optimize space utilization, but this is beyond the scope of this paper.

long strings). The reason is that we can always keep the transferred strings and retrieved IDs in the same sequence (e.g., by array indexes) so that the <term, ID> pair can be easily used to build the local dictionary as described in the *Step 3*.

This method is easy to implement and experimental results presented in [25] have shown that it has achieved higher throughput than current methods in the literature (e.g., [28] and [29]). Moreover, such method is more flexible for various semantic application scenarios, such as transactional data processing and incremental updates, etc.

3.1.2 Primary Index

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 [30] to decompose the local data into multiple parts. Triples in [30] 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 [30], 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.

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. Each of these table uses the part before '$\rightarrow$' as the key, and the part after '$\rightarrow$' as a set of values. For example, the upper 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 only 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.

An alternative implementation where data would be to locally partition by subject. In practice, this would be problematic, since we would either need to have some bound subjects in the query (which is not true at least for our benchmark queries), or perform a lookup on all tables, which would be costly. That said, a partitioning based on subject would be superior for SPARQL DESCRIBE queries, although they are not relevant for our system.

Our work relies on being able to retrieve bindings for single triple patterns very quickly. For most SPARQL queries (and at least all queries in our benchmark), triple patterns have bound predicates and, in most cases, unbound subjects and objects. Following the indexing scheme mentioned above, our system is able to very quickly (in constant time) retrieve binding when the predicate is known, or when the subject and predicate is known, or when the object and predicate in known. This does not limit the generality of the approach, since any other index combination could be chosen.

As in all RDF stores, there is an element of redundancy in terms of data replication. Our index could consume more

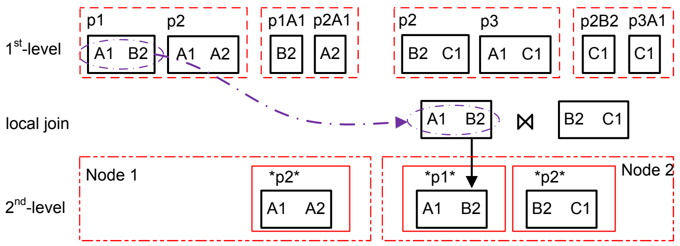


Fig. 4. Query execution and the secondary index building.

space than the vertical partitioning approach in [30], or a compressed index approach such as the one found in [9]. 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.

3.2 Querying

3.2.1 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 of Query(a) in Figure 2, the same as the similar-size partitioning method, 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 [10]. 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.

3.2.2 Secondary Indexes

The local lookup for each triple pattern at each node is very fast, in terms of runtime. The reason is that we only need to locate the corresponding index table in l_1, and then retrieve all the elements. For example, 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 closely related to the execution plan of a query. As SPARQL allows syntactic shortcuts to simplify query formulation, each (conjunctive) query can be parsed and expanded into a set of triple patterns [31]. If a query contains multiple triple patterns, then we will have to perform a join on the common variables (for each pattern). In this case, an query execution plan represented as a tree of triple patterns will be constructed. This tree can be evaluated in multiple ways, but, in our case, we consider (parallel) bottom up evaluation[3].

Based on that, the detailed process on building our secondary indexes is presented in Algorithm 1. We have a queue of queries $\mathbb{Q}$. For each query Q, we assume that we have a planning method (which is beyond the scope of this paper) and it has generated an execution tree with root r already. 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 fast: the index is constructed by a simple *copy* of the redistributed data resulting from a *join* of a query. Namely, it is a byproduct of query execution. Regardless, this index is effective in improving query performance, especially in an iterative analysis environment, because it can be re-used 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 [32]. This means that indexes can be re-used for any

3. Various strategies, based on statistics, heuristics or sampled execution, can be applied for optimizing the join order, but this goes beyond the scope of this paper. In our evaluation in Section 4, we use the query plans generated by RDF-3X [9].

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 initial-
ized as $\emptyset$ at each node, $r$ the intermediate results to be
joined initialized as $\emptyset$.

Main procedure:
1: for each $Q \in \mathbb{Q}$ 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 $r_i \neq null$ 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
```

query containing them and the consequent cost is arises only from local join re-computation.

3.2.3 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, then 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 similar-size partitioning, based on the query loads. Therefore, our approach 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 4. In addition, the theoretical results from [12] can be applied for our approach as well.

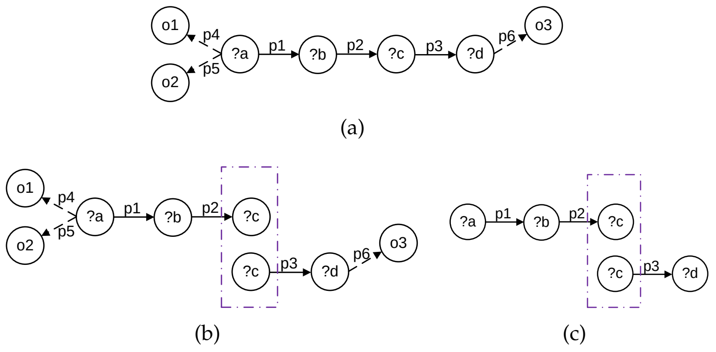


Fig. 5. A complex SPARQL query graph and the *join* in its *full* and *main* path.

3.3 Distributed Filters

Secondary indexes l_k reduce 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. In the meantime, the higher level an index is, the larger its size could be. There are two main reasons for this: (1) l_{k+1} has more tables than l_k, as its properties are constructed by combining the properties of l_k with other indexes'; and (2) the size of a table in l_{k+1} could be larger than that in l_k, for example l_{k+1} could include the *cartesian product* of elements in l_k.

Various strategies can be applied to reduce the size of l, such as reuse of repeated parts between each index, building indexes for frequent graph patterns, etc. [33]. Orthogonally to these approaches, we introduce *distributed filters* as a compact alternative to secondary indexes.

Because some elements taking part in a join could not possibly have a contribution to the output, if we know their join results (for example as intermediate results from a previous query), we can remove such elements before performing the join. Filters operate by filtering out intermediate results at the source, based on aggregate results stored during previous query executions and are focused on a join point (e.g., the join on $?c$ in Figure 5(a)), or, otherwise expressed, an operator on the query plan.

For a join on v between two graph patterns of a query s_i and s_j, we propose one of the following: (1) *full path filters*, which contain all values of v after the join of s_i and s_j, and (2) *main path filters*, containing the results of v over the join between the main path graph of s_i and s_j. The main path graph is defined as the graph consisting only of the triples with a single constant (most commonly the predicate). For example, the query graph in Figure 5(a), can be evaluated by a join over a l_3 and l_2 index on the variable $?c$. In this case, the join results of $?c$ in Figure 5(b) can be used instead of an l_4 table, constituting a 4th-level filter f_4. Bindings for c are filtered at source according to this filter, *for this particular query or any query that subsumes this query*. Figure 5(c) shows the structure of a main path filter for the same query.

These two filters have their own advantages and shortcomings: (1) the full path filter can remove more redundant elements. It is also a byproduct of query execution and can be applied to our approach by projecting out the desired variables and eliminating duplicates. However, the usage of such filter is limited on queries that subsume the entire pattern; (2) the main path filter can be used in more queries, especially since it corresponds to the less selective part of the query, essentially capturing the structure of the graph. Regardless, like many other path filters [34], it can only remove part of the redundant elements, since it is less selective than the query. In addition, it needs pre-computation (or a less efficient query plan starting from the non-selective part).

For a specified query, obviously, the size of a filter will be much smaller than that of the corresponding indexes, because we only store the (discrete) results for the join variable. In the meantime, we note that a filter is less efficient than an index, as it only *reduces* the network communication, while the index can *remove* such communication. In addition, the filtering operation bears some, relatively small, computational cost. We will compare the performance of both techniques in our evaluation in Section 4. In fact, we can adopt a **hybrid** construction for the high-level index, e.g., using index architectures for some join patterns while applying filter architectures for others, so as to achieve the best balancing between performance and space. In addition, for full path filters, it is easy and computationally inexpensive to *reduce* an index to a filter. Finally, we should note that filters are partitioned in the same way as secondary indexes across the network, placing an equal space burden on all nodes.

4 Evaluation

We present an experimental evaluation of our approach to determine the performance of our lightweight indexing, the secondary indexes and the distributed filters. We run the LUBM [18] benchmark over a commodity cluster and compare loading speed and performance of query execution with a top-performing RDF store running on a single node as well as a cluster RDF store.

4.1 Platform

Each computation unit of our cluster is an iDataPlex node with 2 Intel Xeon X5679 processors each with 6 hardware cores running at 2.93 GHz, resulting in a total of 12 cores per physical node. Each node has 128GB of RAM and a single 1TB SATA hard-drive. Nodes are connected by Gigabit Ethernet switch. The operating system is Linux kernel version 2.6.32-220 and the software stack consists of X10 version 2.3 and gcc version 4.4.6.

4.2 Setup

Although in this paper we are focusing on an indexing method, as opposed to a full clustered RDF store, we have performed comparisons on query execution with RDF-3X [9] and 4store [19]. The former represents the state-of-the-art in terms of single machine stores while the latter is a clustered RDF store designed to operate mainly in memory[4]. We have modified the setup so as to isolate the BGP processing costs and nullify, to the extent possible, the advantage of our approach and 4store regarding I/O performance. Specifically, we do not count the time spent on *query parsing*, *plan generation*, *dictionary lookup* or *result output*, so as to focus on analyzing the core performance of query execution. More exactly, we only report times for the operations of *index location*, *index scanning* and the relative *joins* in the execution phase. To achieve this:

- For RDF-3X, we deployed version 0.3.7 on a single node[5]. We add *profiling counters* to the source code.
- For 4store, we installed the latest version 1.1.5 on 16 nodes. We use the default indexes and set the value of the system parameter *segment* to 256, which is the recommended value. In the meantime, we also set *soft-limit* to -1, so that we can retrieve all the results. As 4store provides the desired profiling tools[6], we can directly get the time taken to locate the results and perform the joins on the backend system (storage layer) through examining the *bind time*. We have confirmed this with the 4store community.

We do not compare with MapReduce-based approaches since, due to platform overhead, they do not execute interactive queries in reasonable time. For example, SHARD [11] and, recently, H2RDF+ [35], has runtimes for LUBM (e.g., Q2 and Q9 described below) in the hundreds of seconds.

4.3 Benchmark

Our experiments are based on LUBM benchmark [18]. This benchmark features an ontology for the university domain, synthetic OWL and RDF data scalable to an arbitrary size, by controlling the number of university entities. Meanwhile, it also contains fourteen extensional queries representing a variety of properties [12]. This benchmark has been widely used by RDF stores to compare their performance, especially when large datasets, that can be scaled to arbitrary size, are required (e.g., [12], [35], [36]).

We load LUBM(8000), containing about 1.1 billion triples (about 190GB) and run all 14 queries on this data. As our implementation does not support RDF inference, we use a modified query set to get results for most queries[7]. For example, since the basic graph pattern <?x type Student> returns no results in Query 10, we use <?x type GraduateStudent> instead.

4. Refer to http://4store.org/trac/wiki/Tuning

5. Note that we use single-node RDF-3X as this open-source triple store is commonly used as a performance reference for RDF stores, including clustered solutions, such as [12], [35], etc.

6. Other commercial clustered RDF stores have licensing restrictions and/or do not provide the required profiling functions, such as Virtuoso [15], etc.

7. The rewritten queries can be found at: https://github.com/longcheng11/rdf_framework.

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 approach	Read from disk: 103 Triple encoding: 254 Building l_1: (P, PO, PS) 86 Total: **443**	2483.1K	155.2K

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 implementation 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.

4.4 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 implementation 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[8], 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 implementations (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.

4.5 General Performance

We execute all LUBM queries using l_1 and l_2, since the number of joins in most queries is small. Although our approach 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).

8. Though 4store is a quad-store and has to index graphs IDs, our method will still be much faster even when we consider an ideally linear condition, in which case 4store will take 7078*3/4=5309 secs. In fact, there is only one graph in the dataset and the consequently overhead is very small.

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

Q.	RDF-3X		4store		Our approach		#
	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.

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 implementation, 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 contain the L_3 operation (as defined previously), hence intermediate results still need redistribution over 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 approach (and 4store) has to do synchronization over a distributed architecture, which has an overhead of about 10 ms; (2) our method 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 on 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 implementation, because we only adopt *hash join* as local joins in our method 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 approach 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[9], in the order of *sec*. Moreover, we can further improve the performance of our implementation 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).

Most important, it should be highlighted that we have parallelised all operations[10], including data loading (also for triple encoding) and data querying (also for index building and filtering). For LUBM, our approach is at least an order of magnitude faster at loading data while still keeping query response time within an interactive range.

In practical terms, whether the decreased performance is worth the penalty in executing cheap queries depends on the workload. When the ratio of the volume of data loaded to the volume of data read over the lifetime of the deployment is not very low, the faster loading speed is probably the most important performance metric. Example workloads where this would happen would be interactive, ad-hoc analysis of large datasets and Extract-Transform-Load (ETL) workflows. On the other hand, for deployments service many small requests on an ongoing basis, such as Web endpoints or database backends dominated by reads, our approach would not be ideal.

4.6 Indexes and Filters

We examine the time cost to build indexes and filters, and examine query performance on executing Q2 and Q9, which are the most complex queries. We first build the second-level and third-level index for these two queries and then replace the third-level index by either the *main path* or the *full path* filter.

Figure 6 shows that building a 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. Figure 6 also demonstrates that, with a filter, query execution time is higher than with l_3 (because we still have to redistribute elements over

9. For 4store: (i) for Q2, the entire query time is 335 seconds; and (ii) for Q9, we only provide results when running without system parameter *soft-limit*. If set this parameter to -1, the execution time is more than 7 hours.

10. Note that our method can be implemented in many programming languages, such as MPI or C++ (we have implemented our approach using X10 language [37] and complied it into C++ in this work).

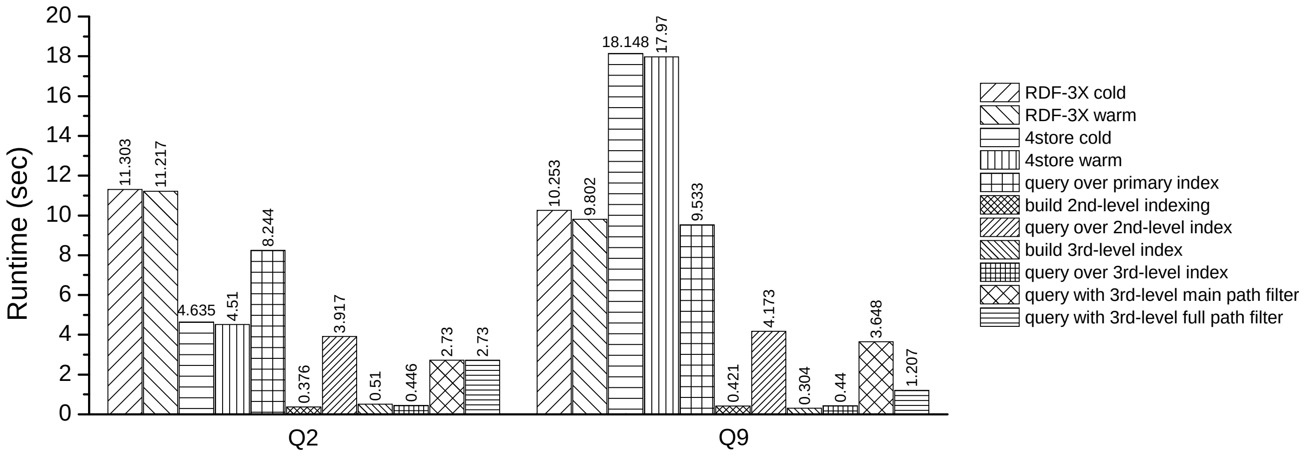
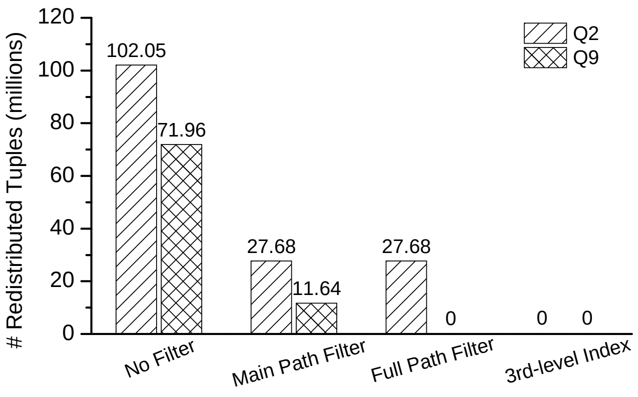
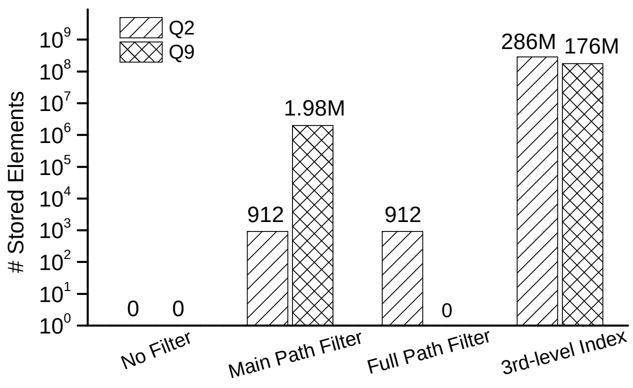


Fig. 6. Runtime for RDF-3X and 4store, and detailed runtime of each implementation for our approach (over Q2 and Q9 using 192 cores).



(a) Number of redistributed tuples



(b) Number of elements in a filter (index)

Fig. 7. Network communication over the index l_2 and the number of elements in a 3rd-level filter on the basis of the 2nd-level index.

the network), but faster than with l_2 (speedups from 1.14 to 3.45) and the other systems, showing a trade-off between space and performance.

To further investigate the effect of filters, we record the total number of received elements and compare to l_2 without a filter. The results are presented in Figure 7(a), showing that network communication can be greatly reduced through filters - a finding further supported by the performance improvement evident in Figure 6. With a *main path filter*, about 70% of data movement associated with Q2 is eliminated. However, we observe that the transfer time is reduced by only 37%, the reason being the communication overhead and the computational cost of filtering. Figure 7 shows that a *full path filter* can sometimes be better than a *main path filter*. For example, network communication is 0 when Q9 is using a *full path filter*, much less than that of the *main path filter*. In contrast, both filters perform the same for Q2. For a full l_3 index, network communication is zero. In Figure 7(b), we show the space overhead for each option (expressed as elements, represented by long integers). We see that, for either method, the cost is much smaller than the number of tuples transferred without a filter (a maximum of 2 million elements stored compared to a minimum of 72 million triples transferred). For comparison, we also include the space cost for l_3 (it can be seen that the size of a filter is up to 1.2% of the index size).

TABLE 3
Number of received tuples at each core (millions) for 192 cores

# received elements	L1		L2	
	Max.	Avg.	Max.	Avg.
Q2	0.987	0.871	0.801	0.532
Q9	1.595	1.593	0.377	0.375

TABLE 4
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

4.7 Load Balancing and Scalability

Because data skew is very common in RDF data [38], we measure load distribution across nodes on Q2 and Q9. We execute both queries over the primary index using 192 cores by recording the number of received elements on each core. As shown in Figure 3, for the two redistributed operations in each query, there is nearly no skew in Q9. In contrast, there exists obvious skew in Q2, which indicates that skew-handling techniques such as the ones in [13], [39] can be applied in our approach to further improve the performance for such queries.

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 Figure 4. 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.

5 Related Work

We position our work against related work in batch processing oriented RDF systems, (clustered) triple stores, graph partitioning based systems and other literature from the database community.

RDF processing systems geared towards batch processing [10], [13], [35] 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 [13] can process massive datasets with zero loading time, its minimum runtime is in minutes, not seconds.

Systems such as SHARD [11] and the one in [40] generally adopt hash-based partitioning techniques. This leads to slower loading of RDF data, e.g., around 30 minutes 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 only L_1 operations. 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 [15], BigData [41], YARS2 [14] and 4store [19] distribute indexes (typically SPO, POS, etc.) 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], [42], [43], [44], [36], 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 [43] 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 approach 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.

Database cracking [45], [46] is an adaptive indexing technique that incorporates continuous self-organization of data storage based on selections in incoming queries. This idea has influenced the design of the secondary index used in the system described here. However, research on cracking is concentrated on incrementally sorting the raw data on a single machine, so as to reduce data lookup time. In comparison, we focus on reducing network communication and apply some concepts behind cracking to distributed systems and RDF data. Additionally, as data in our indexes is unordered, we can also apply the existing cracking methods to our local index, so as to further improve the final query performance of our approach.

Result recycling refers to re-using intermediate results from past query executions. We apply a similar approach to [33], examining caching in a column store using an operator-at-a-time architecture. The main differences with our approach is that (1) we apply this on a distributed setting, (2) we store the remote data rather than the materialization of the intermediate results (i.e., we re-execute the joins locally) and (3) we apply this on RDF data, using indexing structures with different characteristics. In fact, the eviction and retention strategies in [33] could be adapted to our implementation.

Path-based filters proposed for semi-structured data [47] and RDF data [34] can efficiently identify and then reduce elements participating in joins by pre-joining sub-graphs. The two filters proposed here are similar to those in [34], which filter only for the unique items of a *join*. However, there exist four main differences: (1) the height of filters in [34] is limited, as they pre-join all the possible sub-graphs, which is extremely costly both in terms of time and space. In comparison, our *full path filter* is a byproduct of query processing and filters are only built for the specified *join point*; (2) [34] applies filters on a single machine and focuses on techniques to reduce the size of filters while we use partitioned filters in a distributed system and have much less pressure in terms of space. We can incorporate the techniques in [34] to reduce the size of local filters; (3) filters in [34] are mainly used to reduce the time to lookup underlying sorted data, while we focus on reducing inter-machine communication; and (4) the structure of filters in [34] is similar as our *main path filter*, which can be less efficient than our *full path filter* for some queries, as shown in our experiments.

A family of compression techniques have been proposed to reduce the size of RDF data [28], [29], [48], [49], [50], [51], [52]. Some of these approaches, such as HDT [50], facilitate loaded by pre-partitioning data, or by doing the dictionary encoding in advance. In this work, our primary focus is on providing very fast loading and quick retrieval (so as to recover some of the runtime costs from a looser data organization, compared to competing systems). In this light, we are using hash-tables, which are much faster to construct and to perform lookups on, though much less space efficient than approaches such as HDT. A very interesting stream of future work remains on how a more compact representation could be combined with our approach, especially in light of loading data that already has some useful organisation (such as in HDT).

6 Conclusion

In this work, we present a scale-out RDF data processing method designed for fast loading and querying over large-scale data. Based on a simple similar-size data partitioning infrastructure, we propose a dynamic two-tier index architecture and introduce the design of two performance-enhancing distributed filters. Our implementation is evaluated using the LUBM benchmark [18] and the experimental results demonstrate that our approach can load data much faster than a clustered store operating in RAM while remaining within an interactive range for query processing. In fact, our approach can even outperform current systems for some types of queries.

We will investigate further extensions to our design through the application of methods for skew handling (e.g., [39], [53]), index size reduction (or index management) and sort-based local joins which should further improve performance. In addition, we will include additional functionality, such as aggregates, so as to be able to run more complex benchmarks. Although our prototype has much lower coordination overhead than systems based on Hadoop, we can reduce it further by limiting the number of nodes involved in cheap operations. Our long term goal is to develop a highly scalable distributed analysis framework for extreme-scale RDF data.

Acknowledgments

Long Cheng is supported by the DFG in projects DIAMOND (Emmy Noether grant KR 4381/1-1) and HAEC (CRC 912). The computations were performed on the High-performance Systems Research Cluster at IBM Research Ireland.

References

[1] G. Kobilarov, T. Scott, Y. Raimond, S. Oliver, and C. e. Sizemore, "Media meets semantic web - how the BBC uses DBpedia and linked data to make connections," in *Proc. 6th European Semantic Web Conf.*, 2009, pp. 723–737.

[2] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data-the story so far," *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, pp. 205–227, 2009.

[3] W3C, http://www.w3.org/RDF/.

[4] S. Groppe, *Data Management and Query Processing in Semantic Web Databases*. Springer, 2011.

[5] B. McBride, "Jena: Implementing the RDF model and syntax specification." in *SemWeb*, 2001.

[6] J. Broekstra, A. Kampman, and F. Van Harmelen, "Sesame: A generic architecture for storing and querying RDF and RDF schema," in *Proc. 1st Int. Semantic Web Conf.*, 2002, pp. 54–68.

[7] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple indexing for semantic web data management," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 1008–1019, Aug. 2008.

[8] 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*, vol. 18, no. 2, pp. 385–406, Apr. 2009.

[9] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *The VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.

[10] J. Weaver and G. T. Williams, "Scalable RDF query processing on clusters and supercomputers," in *Proc. 5th Int. Workshop Scalable Semantic Web Knowledge Base Systems*, 2009.

[11] 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*, 2010, pp. 4:1–4:5.

[12] J. Huang, D. J. Abadi, and K. Ren, "Scalable SPARQL querying of large RDF graphs," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 1123–1134, 2011.

[13] S. Kotoulas, J. Urbani, P. Boncz, and P. Mika, "Robust runtime optimization and skew-resistant execution of analytical SPARQL queries on PIG," in *Proc. 11th Int. Semantic Web Conf.*, 2012, pp. 247–262.

[14] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "YARS2: A federated repository for querying graph structured data from the web," in *Proc. 6th Int. Semantic Web Conf.*, 2007, pp. 211–224.

[15] O. Erling and I. Mikhailov, "Virtuoso: RDF support in a native RDBMS," in *Semantic Web Information Management*, 2010, pp. 501–519.

[16] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov, "OWLIM: A family of scalable semantic repositories," *Semantic Web*, vol. 2, no. 1, pp. 33–42, 2011.

[17] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "High throughput indexing for large-scale semantic web data," in *Proc. 30th Annual ACM Symp. Applied Computing*, 2015, pp. 416–422.

[18] 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*, vol. 3, no. 2, pp. 158–182, 2005.

[19] S. Harris, N. Lamb, and N. Shadbolt, "4store: The design and implementation of a clustered RDF store," in *Proc. 5th Int. Workshop Scalable Semantic Web Knowledge Base Systems*, 2009, pp. 94–109.

[20] O. Polychroniou, R. Sen, and K. A. Ross, "Track join: distributed joins with minimal network traffic," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2014, pp. 1483–1494.

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

[22] L. Rietveld, R. Hoekstra, S. Schlobach, and C. Guéret, "Structural properties as proxy for semantic relevance in RDF graph sampling," in *Proc. 13th Int. Semantic Web Conf.*, 2014, pp. 81–96.

[23] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann, "DBpedia-a crystallization point for the web of data," *Web Semantics: science, services and agents on the world wide web*, vol. 7, no. 3, pp. 154–165, 2009.

[24] R. Harbi, I. Abdelaziz, P. Kalnis, and N. Mamoulis, "Evaluating SPARQL queries on massive RDF datasets," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1848–1851, 2015.

[25] L. Cheng, A. Malik, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Efficient parallel dictionary encoding for RDF data," in *Proc. 17th Int. Workshop on the Web and Databases*, 2014.

[26] L. Cheng, A. Malik, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Fast compression of large semantic web data using X10," *IEEE Trans. Parallel Distrib. Syst.*, in press, doi: 10.1109/TPDS.2015.2496579.

[27] R. Sedgewick, *Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988.

[28] E. L. Goodman, E. Jimenez, D. Mizell, S. Al-Saffar, B. Adolf, and D. Haglin, "High-performance computing applied to semantic databases," in *Proc. 8th European Semantic Web Conf.*, 2011, pp. 31–45.

[29] J. Urbani, J. Maassen, N. Drost, F. Seinstra, and H. Bal, "Scalable RDF data compression with MapReduce," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 1, pp. 24–39, 2013.

[30] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *Proc. 33rd Int. Conf. Very Large Data Bases*, 2007, pp. 411–422.

[31] T. Neumann and G. Weikum, "RDF-3X: a risc-style engine for RDF," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 647–659, 2008.

[32] J. Umbrich, M. Karnstedt, A. Hogan, and J. X. Parreira, "Hybrid SPARQL queries: fresh vs. fast results," in *Proc. 11th Int. Semantic Web Conf.*, 2012, pp. 608–624.

[33] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves, "An architecture for recycling intermediates in a column-store," *ACM Trans. Database Syst.*, vol. 35, no. 4, p. 24, 2010.

[34] K. Kim, B. Moon, and H.-J. Kim, "R3F: RDF triple filtering method for efficient SPARQL query processing," *World Wide Web*, pp. 1–41, 2013.

[35] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris, "H2RDF+: High-performance distributed joins over large-scale RDF graphs," in *Proc. IEEE Int. Conf. Big Data*, 2013, pp. 255–263.

[36] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing," *Proc. ACM SIGMOD Int. Conf. Management of Data*, pp. 289–300, 2014.

[37] 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*, vol. 40, no. 10, pp. 519–538, 2005.

[38] S. Kotoulas, E. Oren, and F. Van Harmelen, "Mind the data skew: distributed inferencing by speeddating in elastic regions," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 531–540.

[39] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Robust and skew-resistant parallel joins in shared-nothing systems," in *Proc. 23rd ACM Int. Conf. Information and Knowledge Management*, 2014, pp. 1399–1408.

[40] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham, "Heuristics-based query processing for large RDF graphs using cloud computing," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 9, pp. 1312–1327, 2011.

[41] B. Thompson and M. Personick, "Bigdata: the semantic web on an open source cloud," in *Proc. Int. Semantic Web Conf.*, 2009.

[42] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2012, pp. 517–528.

[43] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale RDF data," *Proc. VLDB Endowment*, vol. 6, no. 4, pp. 265–276, 2013.

[44] K. Lee and L. Liu, "Scaling queries over big RDF graphs with semantic hash partitioning," *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 1894–1905, 2013.

[45] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *Proc. Conf. Innovative Data Systems Research*, 2007, pp. 68–78.

[46] S. Idreos, M. L. Kersten, and S. Manegold, "Self-organizing tuple reconstruction in column-stores," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2009, pp. 297–308.

[47] K.-F. Wong, J. X. Yu, and N. Tang, "Answering XML queries using path-based indexes: a survey," *World Wide Web*, vol. 9, no. 3, pp. 277–299, 2006.

[48] J. D. Fernández, C. Gutierrez, and M. A. Martínez-Prieto, "RDF compression: basic approaches," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 1091–1092.

[49] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "TripleBit: a fast and compact system for large scale RDF data," *Proc. VLDB Endowment*, vol. 6, no. 7, pp. 517–528, 2013.

[50] J. D. Fernández, M. A. Martínez-Prieto, and C. Gutierrez, "Compact representation of large RDF data sets for publishing and exchange," in *Proc. 9th Int. Semantic Web Conf.*, 2010, pp. 193–208.

[51] J. M. Giménez-García, J. D. Fernández, and M. A. Martínez-Prieto, "HDT-MR: A scalable solution for RDF compression with HDT and MapReduce," in *Proc. 12th European Semantic Web Conf.*, 2015, pp. 253–268.

[52] H. R. Bazoobandi, S. de Rooij, J. Urbani, A. ten Teije, F. van Harmelen, and H. Bal, "A compact in-memory dictionary for RDF data," in *Proc. 12th European Semantic Web Conf.*, 2015, pp. 205–220.

[53] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Robust and efficient large-large table outer joins on distributed infrastructures," in *Proc. 20th European Conf. Parallel Processing*, 2014, pp. 258–269.

Long Cheng is currently a Post-Doctoral Researcher at TU Dresden, Germany. His research interests mainly include Distributed computing, Large-scale data processing, Data management and Semantic web. He was at organizations such as Huawei Technologies Germany and IBM Research Ireland. He holds a B.E. from Harbin Institute of Technology, China (2007), M.Sc from Universität Duisburg-Essen, Germany (2010) and Ph.D from National University of Ireland Maynooth, Ireland (2014).

Spyros Kotoulas is a Research Scientist at IBM Research Ireland. His research interests lie in data management for semi-structured data, parallel methods for data intensive processing, Semantic Web, Linked Data, reasoning with Web data, flexible data integration methods, stream processing, peer-to-peer and other distributed systems. He holds a BSc (2004) from the University of Crete as well as an MSc (2006) and a PhD (2009), both from the VU University Amsterdam.