WORQ: Workload-Driven RDF Query Processing

Amgad Madkour[1], Ahmed M. Aly[2], and Walid G. Aref[1]

[1] Purdue University, West Lafayette, USA
{amgad,aref}@cs.purdue.edu
[2] Google Inc., Mountain View, USA
aaly@google.com

Abstract. Cloud-based systems provide a rich platform for managing large-scale RDF data. However, the distributed nature of these systems introduces several performance challenges, *e.g.*, disk I/O and network shuffling overhead, especially for RDF queries that involve multiple join operations. To alleviate these challenges, this paper studies the effect of several optimization techniques that enhance the performance of RDF queries. Based on the query workload, reduced sets of intermediate results (or reductions, for short) that are common for certain join pattern(s) are computed. Furthermore, these reductions are not computed beforehand, but are rather computed only for the frequent join patterns in an online fashion using Bloom filters. Rather than caching the final results of each query, we show that caching the reductions allows reusing intermediate results across multiple queries that share the same join patterns. In addition, we introduce an efficient solution for RDF queries with unbound properties. Based on a realization of the proposed optimizations on top of Spark, extensive experimentation using two synthetic benchmarks and a real dataset demonstrates how these optimizations lead to an order of magnitude enhancement in terms of preprocessing, storage, and query performance compared to the state-of-the-art solutions.

Keywords: Intermediate Results · Basic Graph Pattern · Distributed SPARQL Query Processing

1 Introduction

Processing RDF queries involves multiple scans of the same data, *e.g.*, when certain join patterns are frequent and are repeated across multiple queries. This calls for workload-driven mechanisms that cache only the data that is required by the query workload. Network shuffling overhead also degrades query performance in a distributed environment. It occurs when the processing nodes exchange data in order to answer queries. Reducing the network shuffling overhead highly relies on how the data is partitioned across the nodes.

This paper presents *Wo*rkload-driven *R*DF *Q*uery Processing (WORQ, for short), a system that encapsulates several optimizations that significantly enhance the performance of RDF queries. In particular, WORQ addresses three main issues: 1) how to efficiently *partition* the RDF data in an online fashion, 2) how to *reduce* the intermediate join results of an RDF query in an online fashion, and 3) how to *cache* reusable intermediate join results instead of the final results of an RDF query.

Workload-Driven Partitioning: Data partitioning is common in distributed data management systems. The RDF data is typically divided into several partitions, and then is distributed across the cluster machines. The objective of partitioning is to reduce the query execution time by leveraging parallelism. Data partitioning incurs a preprocessing overhead as it needs to be performed over the whole data. However, for a real workload, only a small fraction of the data is accessed (*e.g.*, see [25]). WORQ adopts a workload-driven approach when partitioning the data. For each query, WORQ identifies each query triple (*i.e.*, an entry consisting of bound and unbound subject, property, and an object) as a subquery. Then, WORQ partitions the data triples by the join attribute of each subquery. The join attribute represents the variable that connects two or more query triples. The join can be between subjects, properties, objects, or a combination of the three attributes. WORQ partitions the data only once for every new query join pattern that is identified.

Join Reductions: Tables are one way of storing RDF data triples. When a single query involves joins between multiple tables that correspond to different query patterns, every binary join operation generates intermediate join results (or *intermediate results*, for short). The intermediate results represent the data that satisfies the binary join and eventually contributes to the final result of the query. However, intermediate results may contain redundant data triples that do not match all the query joins. WORQ minimizes the intermediate results by precomputing join reductions through Bloom-joins [8, 16].

Caching: To boost query performance, caching can be employed to improve query response time and increase the throughput of execution. One caching approach is to cache the *results* of each query. However, caching the unique query results incurs significant memory storage overhead. In contrast, WORQ caches (in main memory) the join reductions that correspond to the frequent join patterns. These reductions can be reused by other queries that share the same query patterns.

Queries with Unbound Properties: Some query workloads may have query triples with unbound (*i.e.*, unspecified) properties. For example, the query triple `:John ?x :Mary` queries all data triples that have a subject `:John` and an object `:Mary`, where `?x` specifies an unbound property. Answering unbound property queries is challenging for RDF systems that adopt a specific RDF partitioning scheme. Assuming that the data is vertically partitioned [1, 13] (VP, for short), the data triples are split into separate files denoted by the property (*i.e.*, predicate) name, where each file contains the subject and object representing the property. Using VP, answering unbound property is challenging because all property files need to be accessed or an index needs to be built on top of each file. In contrast, WORQ utilizes Bloom filters as indexes to efficiently answer unbound property queries.

WORQ is implemented as part of the Knowledge Cubes (KC) proposal [17]. The source code [3] for a Spark-based implementation of WORQ is publicly available for download. Our experimental setup includes two synthetic benchmarks, namely WatDiv [4] and LUBM [10], and a real dataset, namely YAGO2s [7, 12]. The purpose of the experiments is to demonstrate three aspects of WORQ : 1) the preprocessing time required given an RDF dataset, 2) the storage overhead incurred to create the RDF database, and 3) the query processing time when answering RDF queries with respect

[3] http://github.com/amgadmadkour/knowledgecubes

to partitioning and caching. The results illustrate how the presented optimizations provide at least an order of magnitude better results on the three aforementioned aspects when compared to the Hadoop-based state-of-the-art solution.

The contributions of this paper can be summarized as follows:

- We present workload-driven partitioning of RDF triples that can join together in order to minimize the network shuffling overhead based on the query workload.
- We present the use of Bloom filters for computing RDF join reductions online.
- Rather than caching the results of an RDF query, we show that caching the RDF join reductions can boost the query performance while keeping the cache size minimal.
- We study an efficient technique for answering RDF queries with unbound properties using Bloom filters.

The rest of this paper proceeds as follows. Section 2 presents the online reduction of RDF data. Section 3 presents workload-driven partitioning in WORQ . Section 4 presents how WORQ answers unbound-property queries. Section 5 presents the experiments performed over the WatDiv, LUBM, and YAGO datasets. Section 6 presents the related work. Finally, section 7 presents concluding remarks.

2 Online Reduction of RDF Data

WORQ employs Bloom-join [8, 16] to compute the reductions between vertical partitions. Many cloud-based systems [13] use vertical partitioning (VP) [1] including the state-of-the-art [27]. VPs can be realized over any relational database system and stored in cloud data sources (e.g. Parquet, ORC2). Bloom-join determines if an entry in one partition qualifies a join condition with another partition. The reductions can be computed in an online fashion using Bloom-join instead of precomputing all possible reductions in an offline fashion (*i.e.*, during the preprocessing phase [27]). Bloom-join utilizes a probabilistic data structure, termed Bloom filter [8]. A Bloom filter does not physically store items, but rather hashes the input against different hash functions. The main functionality of a Bloom filter is to determine the existence of an item. Bloom filters can have false-positives, but no false-negatives. Bloom filters are fast to create, fast to probe, and small to store. Also, the false-positives introduce a small percentage of irrelevant rows that eventually are not joined in a Bloom-join. During the evaluation of a join, WORQ uses Bloom filters to probe the join attributes of the query join-patterns. The Bloom filters representing the join attributes filter the rows in both partitions involved in the join, and the results are materialized as a reduction for a specific join pattern, or *reductions*, for short.

Figure 1 gives an example of using Bloom filters to compute a join reduction. The query has a BGP join between `:mention` and `:tweet` on the Subject attribute. WORQ uses the Bloom filter of $\texttt{BloomFilter}^{\text{sub}}(\texttt{:tweet})$ to compute a reduction for the `:mention` property on the subject column. `:tweet`'s Bloom filter consists of the elements `:John`, `:Mike`, and `:Alex`. Each element in the subject column of the `:mention` partition is probed against the `:tweet` Bloom filter. The reduction for `:mention` represents all the rows that qualify a join between the vertical partitions `:mention` and `:tweet` on the subject attribute. Figure 1 illustrates the entries

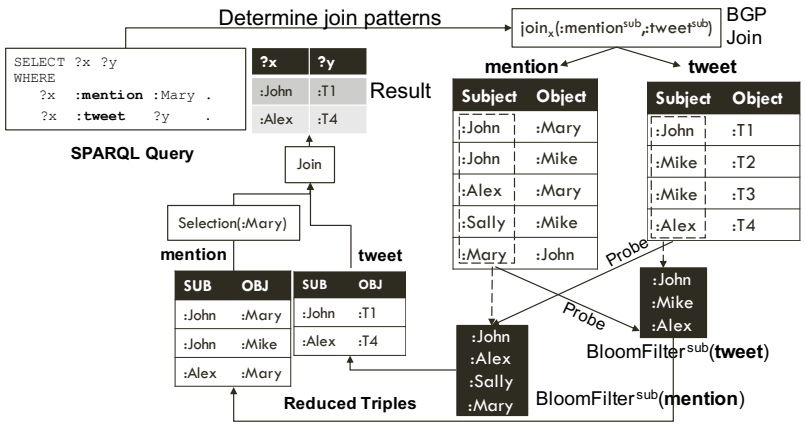

Fig. 1. Evaluating a SPARQL query using Bloom-join between :mention and :tweet

that qualify the join between `:mention` and `:tweet`, where the vertical partition of `:mention` is reduced from five entries to only three qualifying entries. Similarly, the vertical partition of `:tweet` is reduced from four entries to only two qualifying entries. The reductions for both properties are cached by WORQ in order to be reused by other queries that share the same join patterns. In other words, the `:mention` reduction can be reused by the `:mention` property if it joins with `:tweet` on the subject attribute. Also, the `:tweet` reduction can be reused by the `:tweet` property if `:tweet` joins with the `:mention` property on the subject attribute.

WORQ does not apply selection (*i.e.*, filtering) operations on the original data triples (*i.e.*, VP). Instead, selections are applied on the reductions after the reductions are computed. For example, the reduction for `:mention` contains a selection on the object, namely `:Mary`. However, the selection has been delayed until the reductions have been computed from the original data triples. The advantage of delaying the selection is that the reductions can be reused by other queries that share the same join patterns. However, if selections are pushed early on the original data triples, then the reductions will not be representative of the join operation between the query triples. Finally, the resulting reductions (including the ones that have been filtered) are joined together based on the join attribute indicated by the query triples. WORQ does not require a specific join algorithm to be used. Distributed join algorithms, e.g., broadcast hash join or sort-merge join that are employed by distributed computational frameworks, *e.g.*, Spark, can be used [3]. Figure 1 illustrates the final result of the query after joining both the query triples representing `:mention` (after the selection) and `:tweet` properties, where two entries qualify the join result.

N-ary Join Reductions

WORQ computes the reductions online instead of pre-computing the reductions offline [27]. In addition, WORQ computes the reductions between all the possible (n-ary) query-triples instead of computing the reductions in binary form [27].

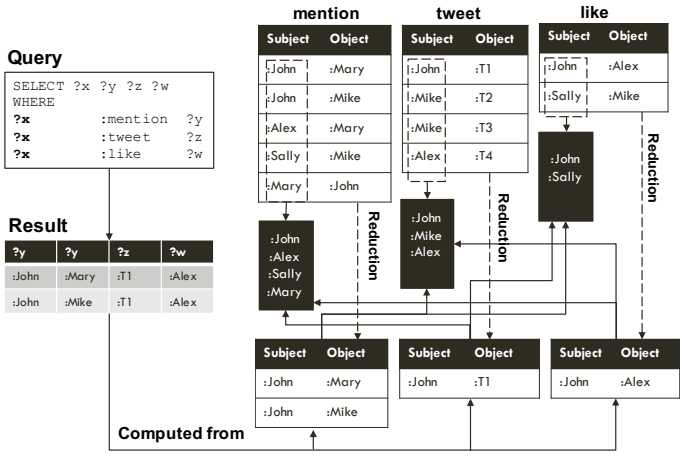

Fig. 2. N-ary join between the reductions of three query triples involving the :mention, :tweet, and :like VPs

Figure 2 illustrates a SPARQL query with three query-triples that share the same join attribute (*i.e.*, variable `?x`). When the join is computed between the `:mention`, `:tweet`, and `:like` VPs, only the data triples that are common amongst the three VPs will qualify as a result. WORQ utilizes Bloom join to reduce the number of data triples in every VP involved in the join operation, and hence reduces the intermediate results between the three join operations. WORQ uses the Bloom filters representing the join columns of the three query triples (the subject Bloom filters, in this instance) to reduce the VP entries to the ones that would qualify the join operation. For example, the `:mention` VP is reduced from five data triples to two triples that have `:John` as the subject because `:John` is the only resource that qualifies the `:tweet` Bloom filter on the subject and the `:like` Bloom filter on the subject. The same applies to the `:tweet` VP, where `:John` is the only resource that qualifies the `:mention` Bloom filter on the subject and the `:like` Bloom filter on the subject. Finally, WORQ uses the computed reductions instead of the VPs to evaluate the query. The result of the query includes two rows corresponding to the only resource common across the three property-VPs. The computed reductions are cached to be reused by any other query that contains a join between the three properties on the subject attribute.

Caching of Reductions

Rather than caching portions of the original RDF data or the final query results, WORQ caches (in main-memory) the *reductions* that correspond to the join patterns that are discovered during query processing. Caching intermediate results (*i.e.*, reductions) is suitable in situations where the query workload consists of a high number of unique queries that share similar patterns. In contrast, caching the results is suitable in situations where the query workload consists of a high number of frequent queries that do not necessarily share the same query pattern. WORQ is suitable for the former case

where many unique queries can utilize the reductions without the need to cache all their results. WORQ does not assume a specific cache-eviction policy, *i.e.*, any eviction policy. WORQ employs least recently used (LRU) strategy where evicted reductions can be saved to disk and be reused if the pattern they represent reoccurs. Also, the advantage of saving to disk is that filtering will not be performed again. The cache-eviction policy is beyond the scope of this paper.

3 Workload-Driven Partitioning

Rather than relying on a predefined partitioning criteria (*e.g.*, using the subject only), WORQ partitions the RDF data according to the join patterns in the queries received so far. WORQ aims at placing the partitions of the reductions that share the same join attribute on the same machine, which minimizes the shuffling overhead, and more importantly, reduces the query response time. Instead of partitioning the VP, WORQ partitions the *reduction rows* across the machines. After a query is parsed, WORQ identifies the join attributes in the query. Based on the join attributes, the reductions that need to be partitioned are determined. Reduction partitioning is performed only once, and the resulting partitions are reused by any query that has the join pattern that corresponds to the reduction.

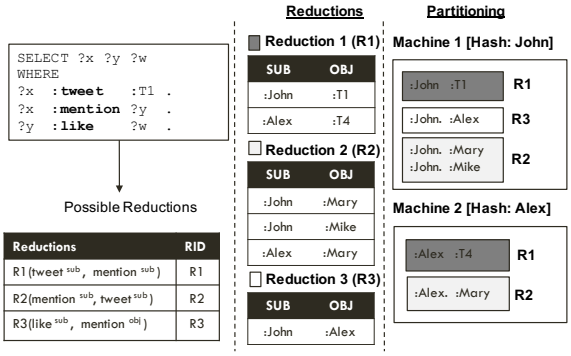

Fig. 3. Workflow for workload-driven partitioning

Figure 3 illustrates a set of query join patterns and their corresponding reductions. The join pattern representing the `:tweet` property uses the reduction denoted by `R1` on the subject. The join pattern representing the `:like` property uses the reduction denoted by `R3` on the subject as well. WORQ partitions the rows of every reduction based on the join attribute (*i.e.*, the subject or object). In Figure 3, the reductions representing `R1,R2,R3` are partitioned using the subject (as the reductions are based on the subject attribute). The reduction rows are hash-distributed across the machines using the join attribute (*i.e.*, subject or object). This partitioning scheme guarantees that all the data triples that are related to the join attributes of the query are co-located on the same machine, and thus allowing the reductions to be computed locally.

4 Queries with Unbound Properties

The performance of unbound-property queries depends on the adopted RDF partitioning scheme. If the data is vertically partitioned, answering unbound-property queries becomes challenging because all the VPs need to be iterated. A straightforward approach to query the unbound properties in a distributed setting is to store the RDF data triples in a single file (*i.e.*, triples file). Distributed file systems, *e.g.*, HDFS, split the files into a set of blocks and distribute the blocks across machines. In this case, RDF query processors can evaluate unbound-property RDF queries in parallel [26], where each machine processes a set of blocks. We refer to this baseline approach as *RDF-Table*. We implement this baseline for evaluation purposes.

WORQ utilizes Bloom filters as cheap indexes to efficiently answer unbound-property queries over data that has been vertically partitioned. WORQ performs two steps to determine the matching properties. The first step is called the *identification* step, where a set of candidate properties are identified. The second step is called the *verification* step, where the candidate properties are verified to eliminate the possibility of false-positives. Given a query, WORQ uses the existing Bloom filters to discover the unbound property. WORQ relies on the *bound* attributes (*i.e.*, subject and object) to discover the matching properties.

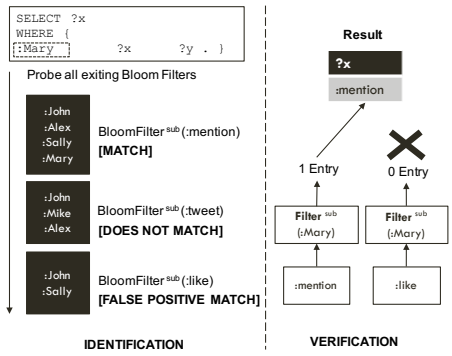

Fig. 4. The identification and verification steps to answer unbound-property queries

Figure 4 illustrates the identification step for answering unbound-property queries. First, the unbound and bound attributes are identified. Then, the bound attributes are used to probe the Bloom filters to determine if the bound values exist for a specific property. If a value exists, the corresponding property is added as a candidate for answering the query. For instance, in Figure 4, `:Mary` exists in the `:mention` property, and is found using a *MATCH* in the corresponding Bloom filter. However, `:Mary` does not exist in the `:tweet` property, and hence the Bloom filter returns *DOES NOT MATCH*. Although `:Mary` does not exist in `:like`, the Bloom filter returns a *MATCH*, which is a false positive.

Given that Bloom filters can incur false positives, a verification step is needed to ensure the correctness of query evaluation. WORQ verifies the candidate properties by issuing a filter based on the bound attributes with the value indicated in the query triple (*i.e.*, the value that made the candidate property match). If the result-set includes at least one match, then WORQ determines that the candidate property was identified correctly. Otherwise, the candidate property is discarded. Disqualifying data will not happen frequently based on the false-positive rate of the constructed Bloom filters.

5 Experiments

WORQ is compared against S2RDF [27], a Spark-based system that runs over Hadoop. S2RDF [27] proposes an extension to VP, namely ExtVP, where reductions of entries are computed for every vertical partition. S2RDF utilizes *semi-join reductions* [6] to reduce the number of rows in a partition. The reductions represent all RDF query combinations that appear in SPARQL queries (*i.e.*, Subject-Subject, Subject-Object, Object-Subject, Object-Object). However, S2RDF exhibits a substantial preprocessing overhead. Semi-joins are expensive to compute, and generate large network-traffic. In addition, S2RDF generates a large number of files to represent the reductions of the original data. S2RDF translates SPARQL queries to SQL and runs them on Spark SQL. S2RDF has outperformed Hadoop-based systems such as H2RDF+, Sempala, PigSPARQL, SHARD, and other systems such as Virtuoso, where S2RDF has achieved (on average) the best query execution performance [27]. Accordingly, this paper presents a comparison with S2RDF only as S2RDF represents the state-of-the-art Hadoop-based RDF query processing system. WORQ is implemented over Spark (v2.1) where it utilizes Spark DataFrames to represent the reductions. WORQ does not translate the query to SQL. Instead, WORQ implements joins as a series of Spark DataFrame joins. To guarantee a fair setup, all Spark-related parameters are unified for both WORQ and S2RDF. The data for both systems is stored using Parquet [4] columnar-store format. Vertical partitioning has been implemented as a baseline.

5.1 Experimental Setup

The experimental setup datasets and queries proposed by Abdelaziz *et al*. [2] are used. Our experiments are conducted using a real dataset (YAGO2s [7, 12]) as well as two synthetic benchmarks (WatDiv [4] and LUBM [10]) that provide widely-adopted query workload generators:
1. **WatDiv** provides a stress-test query workload that allows generating several queries per-pattern. One Billion triples have been generated to demonstrate the query execution performance and preprocessing performance (i.e., the number of files generated, disk space utilization, and loading time). A pre-generated workload provided by WatDiv [4] contains 5000 queries that cover 100 diverse SPARQL patterns, each having 50 variations. A variation represents different bound values for the same query pattern. The variations allow measuring the performance of specific patterns under different

[4] parquet.apache.org

selectivities. For the unbound-property queries, we use the query workload provided by Alvarez-Garcia *et al.* [5] that represents 500 queries covering three combinations namely, unbound subject with bound object, unbound object with bound subject, and bound subject and object.
2. **LUBM** provides a query-workload generator, where 1000 queries are generated. Unlike WatDiv, LUBM does not specify the number of patterns.
3. **YAGO2s** consists of 245 million real RDF triples. YAGO2s benchmark queries are used to compare the query execution time [19, 27]. There is no publicly available real query workload for YAGO. Generating synthetic queries for YAGO is similar to what WatDiv and LUBM provide while they guarantee generating all possible query shapes.

Our experiments are conducted using an HP DL360G9 cluster with Intel Xeon E5-2660 realized over 5 nodes. The cluster uses Cloudera 5.9 consisting of Spark 2 as a computational framework and Hadoop HDFS as a distributed file-system. Each node consists of 32 GB of RAM, and 52 cores. The total HDFS size is 1 Terabyte. The experiments measure various aspects of WORQ including (1) the number of generated files, (2) the filesystem size, (3) the loading time, (4) the workload query execution performance, (5) the overhead of caching results instead of caching reductions, and (6) the execution performance of unbound properties queries. The data for the 3 benchmarks is loaded into memory before execution.

5.2 Experimental Results

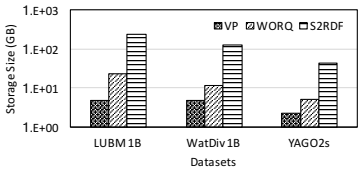

Fig. 5. Disk space utilization

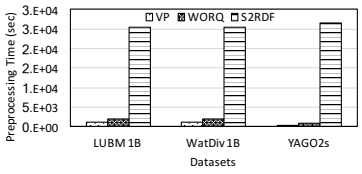

Fig. 6. Preprocessing time

Preprocessing Performance Figure 5 gives the disk storage overhead incurred by the three systems over the LUBM, WatDiv, and YAGO2s datasets. VP introduces minimal space overhead across all three systems. The reason is that VP only needs to partition the original triple file based on the property name. Storage in WORQ is composed of the VP and the Bloom filters. S2RDF precomputes all the possible reductions for binary joins ($O(n^2)$, where n is the number of VPs), and stores them on disk along with the original data. Thus, S2RDF introduces the highest disk storage overhead.

Figure 6 gives the preprocessing time for all three systems over the LUBM, WatDiv, and YAGO2s datasets. VP has the smallest loading time due to its simplicity, followed by WORQ, and then S2RDF. The majority of time spent by S2RDF in the preprocessing time involves creating the proposed partitions called ExtVP. The computation involves

performing semi-joins between binary partitions in a distributed fashion causing high network shuffling overhead. WORQ incurs a minor overhead compared to VP due to the computation of the Bloom filters.

Query Workload Awareness For the remaining experiments, the results of VP are omitted due to its low performance. The following experiments demonstrate the query performance of both WORQ and S2RDF across different aspects, *e.g.*, the total execution time, the mean execution time per query pattern, and the mean execution time given the number of join-triples in a query. WatDiv and LUBM are used due to the availability of workload generators while YAGO2s is omitted as a real query workload is unavailable. However, a set of benchmark queries [27] are used to measure the performance against the YAGO2s dataset. In S2RDF, the partitioning is done for every query and takes place while the queries are being evaluated. S2RDF reports the overall execution time which includes both the partitioning and the actual execution time. WORQ follows the same procedure when reporting the overall execution time.

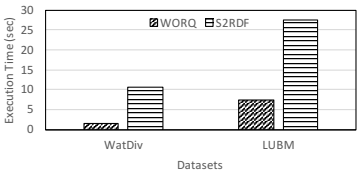

Fig. 7. Mean query execution time

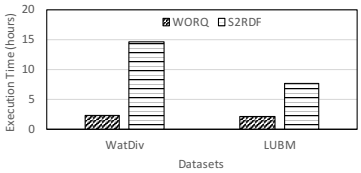

Fig. 8. Total query execution time

Figure 7 and Figure 8 give the mean and total execution times based on executing 5000 queries over WatDiv (1 Billion triples) and 1000 queries over LUBM (1 Billion triples). WORQ is consistently better across the two benchmarks. The difference in performance is attributed to the combination of efficient partitioning and the caching of reduction employed by WORQ as illustrated in later experiments. WORQ reduces the relations to be joined by computing light-weight reductions that can fully represent the original data in answering the RDF queries. Rather than scanning the original (large) data for each query, the light-weight reductions are used instead.

The difference in performance between LUBM and WatDiv is attributed to the characteristics of both benchmarks in terms of the number of properties and the query workload representing each dataset. LUBM consists of 18 properties while WatDiv consists of 86 properties. The 1 Billion triples for LUBM and WatDiv are distributed across 18 and 86 properties, respectively. WORQ performs well with the increase in the number of properties. In real datasets, *e.g.*, YAGO2s [7, 12], the number of properties are in hundreds, making WORQ more appropriate to use than S2RDF.

Figure 9 gives a break-down of the query execution of 5000 queries over WatDiv (1 Billion triples) per query pattern. The x-axis represents the query numbers and the y-axis represents the execution time. A query pattern represents a set of one or more query triples (*i.e.*, BGP triples) that vary based on the bound and unbound attributes,

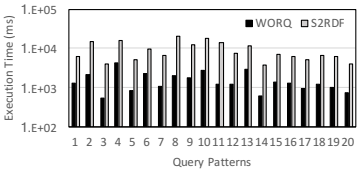

Fig. 9. Mean execution time per query pattern over WatDiv 1 Billion dataset

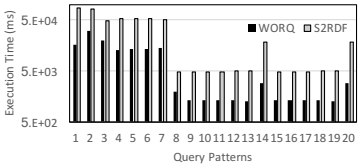

Fig. 10. Mean execution time per query pattern over LUBM 1 Billion dataset

e.g., one pattern can have two query triples joined by the subject attribute while another pattern would be based on two query triples joined on the object attribute. For every pattern, the mean execution time is recorded for the two systems. Figure 9 shows that WORQ executes each pattern nearly an order of magnitude faster than S2RDF.

Figure 10 gives a break-down of executing 1000 queries over LUBM (1 Billion triples per query pattern). Similar to WatDiv, the mean execution time is recorded per pattern for the two systems. The number of patterns included in the LUBM query workload is 20. Figure 10 shows that all the patterns are executed faster by WORQ than S2RDF.

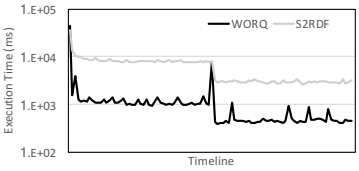

Fig. 11. Execution timeline for two query pattern over WatDiv 1 Billion dataset

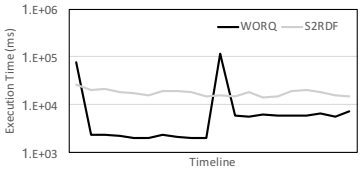

Fig. 12. Execution timeline for two query pattern over LUBM 1 Billion dataset

Figure 11 gives the performance when executing only two patterns over the WatDiv benchmark. The x-axis represents the timeline, where we execute one query pattern first, and then execute another pattern. There are two major spikes in the performance of WORQ that reflect the first time each query pattern was executed. For each pattern, a high query execution overhead is exhibited at the beginning, followed by near-linear performance for the rest of the queries that share the same join pattern.

Figure 12 repeats the same experiment for two patterns over the LUBM benchmark. Similar to Figure 11, the first time a join pattern is executed, a spike in execution time is exhibited followed by a near-linear performance for the remaining queries. Unlike WatDiv, the computation of the query patterns for the first time over LUBM consumes more time than S2RDF. However, the overall execution time of WORQ outperforms S2RDF as Figure 8 illustrates.

We analyze the effect of query triples on the query execution. The WatDiv query workload contains a set of 100 representative patterns and is used for the analysis.

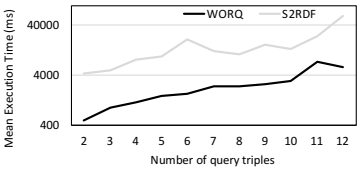

Fig. 13. Mean execution time - Number of triples per query over WatDiv 1 Billion

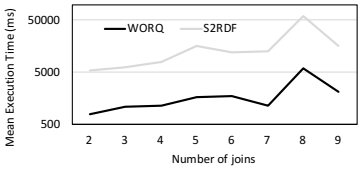

Fig. 14. Mean execution time for joins per pattern over WatDiv 1 Billion

LUBM benchmark is discarded for this experiment as WatDiv provides a workload with more diverse shapes than LUBM.

Figure 13 gives a break-down of executing 5000 queries over WatDiv (1 Billion triples) given the number of triples per query. From the figure, the number of triples affects the overall query performance, where the query execution time increases as more triples are processed.

Figure 14 gives a break-down of the mean query execution time for 5000 queries over WatDiv (1 Billion triples) based on the number of joins between query triples. This is different from the number of query triples experiment, where the number of joins experiment measures the maximum number of joins identified per query, *e.g.*, a query may contain five query triples, but contains a join between two query triples only. To create the experimental setup, every query is first placed in a join group based on the maximum number of joins that it has. Then, the mean execution time is measured for queries within a join group. WORQ achieves nearly an order of magnitude better performance than S2RDF.

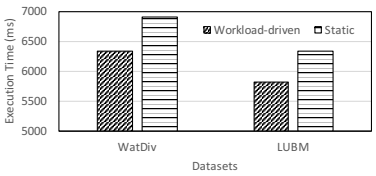

Fig. 15. Mean query execution time using workload-driven and static partitioning

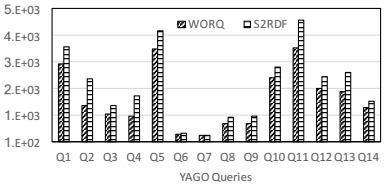

Fig. 16. Execution time of 14 query patterns over YAGO2s dataset

Figure 15 gives a break-down of the mean execution time using workload-driven partitioning and static partitioning of WORQ to illustrate the effect of using the workload-driven component only. Static partitioning is based on subject. In workload-driven partitioning, every query is partitioned based on the join patterns of the query. In contrast, static partitioning is performed based on a pre-specified criteria, *e.g.*, partitioning by subject. Static partitioning was performed on the subject column. Figure 15 demonstrates that workload-driven partitioning contributes positively towards the overall query execution performance over the two datasets. The partitioning time is depen-

dent on where data is originally stored on the cluster and generally incurs a minor cost. The query evaluation time given where data is partitioned dominates the execution time.

Figure 16 gives a break-down of the query execution over 14 benchmark YAGO2s queries [27]. The x-axis represents the query numbers and the y-axis represents the execution time. The queries were designed to take into consideration various query shapes, *e.g.*, star-shaped, and resources selectivities. Each query was executed 5 times using different selective predicates and the average time was reported. For every query, the corresponding reductions for both WORQ and S2RDF were loaded into memory in advance. WORQ achieves better query execution performance over all queries.

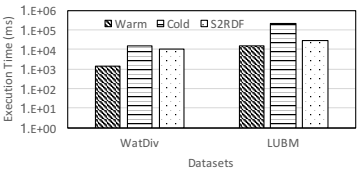

Fig. 17. Mean query execution time given warm and cold cache for WORQ over WatDiv and LUBM

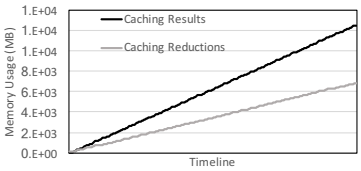

Fig. 18. Memory usage based on caching results and caching reductions

Caching of Reductions Figure 17 demonstrates the effect of caching on the query performance. *Cold* queries are those with patterns that have not been executed before, i.e., that have no corresponding reductions in the cache. *Warm* queries are those that share the same pattern as queries that executed before, i.e., that have corresponding reductions in the cache. The figure gives the mean execution time of 5000 queries from the WatDiv benchmark and 1000 queries from the LUBM benchmark. The figure demonstrate how utilizing cached patterns (*i.e.*, reductions) achieves better query execution performance. The reason LUBM cold cache is worse is because while both datasets are of the same overall size (1B triples), one contains 18 files/predicates (LUBM) in contrast to 87 files/predicates in WatDiv so the filtering time is higher for LUBM queries. Also, WORQ pays a price only once when a query pattern is seen for the first time. However, the cold-start cost is minor.

Figure 18 gives a break-down of the memory usage over 5000 unique queries covering 100 patterns. Using a workload of 5000 unique queries, the figure demonstrates how the size of the cached queries grows over time and surpasses the size of cached reductions. The memory usage for caching the query results can reach more than 10 GB over 5000 queries while caching reductions exhibits a slower memory usage curve. The conclusion is that caching the reductions is more suitable than caching the query results in situations where there are many unique queries that share common patterns.

Performance of Unbound-Property Queries Schatzle *et al.* [27] do not evaluate the performance of S2RDF for unbound-property queries as it is out of the scope of their

current work. In addition, S2RDF adopts a VP structure to answer queries, leading to degraded query performance over unbound-property queries. Therefore, we use the RDF-Table approach described in section 4 as a baseline. We evaluate three query patterns based on the attributes of an RDF triple, namely a bound subject and object, a bound subject, and a bound object.

Table 1. Unbound property results - (BSO) Bound Subject and Object, (BS) Bound Subject, (BO) Bound Object

System	BSO-Mean	BSO-Sum	BS-Mean	BS-Sum	BO-Mean	BO-Sum
WORQ	1.25 ms	10.49 min	4.18 ms	34.84 min	3.52 ms	29.34 min
RDF-Table	5.3 ms	44.44 min	3.80 ms	31.67 min	4.35 ms	36.26 min

Table 1 gives the result of processing 500 queries with bound subject and object (BSO), bound subject (BS), and bound object (BO) over WatDiv (1 Billion triples). For bound subject and object (BSO), the mean execution time per query is nearly five times better than the baseline. This is attributed to the Bloom filter usage, where the number of false-positives is reduced by evaluating the properties against two bound values instead of one bound value, *e.g.*, queries with bound subject only or a bound object only. For bound subject (BS), the mean execution time of WORQ is comparable to that of RDF-Table. This performance is due to two main reasons. The first is the efficiency of RDF-Table within Spark as RDF-Table performs predicate pushdown filtering in parallel and the result is aggregated back to the driver (*i.e.*, master node). The second is that the data of the RDF-Table is sorted by the subject, allowing the predicate-pushdown to work efficiently. For bound object, the mean execution time is also comparable to that of RDF-Table. The overall execution time of WORQ is better than RDF-Table. The reason for the better result is attributed to the lack of sorting on the object column for the RDF dataset. This gives WORQ performance advantage when executing bound object queries.

6 Related Work

Graph-based partitioning is an NP-complete problem [14], and hence hash partitioning heuristics [21, 31] are employed instead of graph-based partitioning in order to partition RDF data efficiently. However, sophisticated partitioning techniques [11,15,22,28] cannot guarantee that no data will be shuffled when processing complex queries with multiple joins. Several techniques [23, 29] utilize the query workload to enhance the partitioning of RDF data. In addition, one study [4] demonstrates the need to continuously adapt to workloads in order to guarantee consistent performance. Characteristic sets [18] capture the set of properties that occur together for a given subject. However, characteristic sets are data-driven and are tied to star-shaped queries only. Castilo *et al.* [9] perform evaluation of SPARQL queries using (offline) materialized results coined RDFMatView indexes. In contrast to materialized views, WORQ does not materialize results but instead identifies reductions that can be reused across queries that

share the same join patterns. H2RDF+ [20] provides a result-based workload-aware RDF caching engine that manages to dynamically index frequent workload subgraphs in real time. However, caching the final results of RDF queries incurs significant storage overhead and cannot generalize to a broader query workloads. Yang *et al*. [30] propose caching the intermediate results of basic graph patterns in SPARQL queries. However, the proposed approach is tied to the join orders that would result in different intermediate results. Alvarez-Garcia *et al*. [5] introduce a compressed index called k2-triples for answering unbound-property queries. However, the proposed index is not applicable in a distributed setting. Ravindra *et al*. [24] uses a non-relational algebra based on a TripleGroup data model to answer unbound-property queries.

7 Concluding Remarks

This paper presents several optimizations for RDF query processing over vertically partitioned triples. First, we present how to use Bloom join to compute reduced sets of intermediate results (or reductions, for short) that are common for certain join pattern(s) in an online fashion. Second, we study the effect of caching these reductions instead of caching the final results of each query. Third, we present how to partition the RDF data triples using the join attributes of the query instead of using a predefined partitioning criteria. Fourth, we present how to efficiently answer queries with unbound properties using Bloom filters. Extensive experimentation using the WatDiv, LUBM, and YAGO2s demonstrate how a realization of these optimizations can lead to an order of magnitude enhancement in terms of preprocessing time, storage, and query performance. Bloom filters/join is one case study. N-ary filtering can utilize any set membership structure (*e.g*., Bloom, Cuckoo, Roaring Bitmaps) so long as we can add and check elements in a set. The novelty is in how membership structures (*e.g*., Bloom Filter) are used to filter data and answer unbound property queries efficiently in a distributed setting. For future work, we will investigate further query processing enhancements including load-balanced partitioning of reductions, generalized filtering (exact vs. approximate structures), and spatio-temporal RDF filtering.

References

1. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic Web data management using vertical partitioning. *VLDB*, 2007.
2. I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. *PVLDB*, 10(13):2049–2060, 9 2017.
3. G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis, and D. Plexousakis. RDF Query Answering Using Apache Spark : Review and Assessment. *DESWEB*, 2018.
4. G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified Stress Testing of RDF Data Management Systems. *ISWC*, pages 197–212, 2014.
5. S. Álvarez-García, N. Brisaboa, J. D. Fernández, M. A. Martínez-Prieto, and G. Navarro. Compressed vertical partitioning for efficient RDF management. *Knowledge and Information Systems*, 44(2):439–474, 8 2015.
6. P. a. Bernstein and D.-M. W. Chiu. Using Semi-Joins to Solve Relational Queries. *Journal of the ACM*, pages 25–40, 1981.

7. J. Biega, E. Kuzey, and F. M. Suchanek. Inside YAGO2s. In *WWW*, pages 325–328, New York, New York, USA, 2013. ACM Press.
8. B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
9. R. Castillo and U. Leser. Selecting materialized views for RDF data. In *ICWE*, 2010.
10. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics*, pages 158–182, 2005.
11. S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing. In *SIGMOD*, pages 289–300, 2014.
12. J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence*, 194:28–61, 1 2013.
13. Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *VLDB Journal*, 2015.
14. G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM*, pages 359–392, 1998.
15. K. Lee and L. Liu. Scaling queries over big RDF graphs with semantic hash partitioning. *VLDB*, pages 1894–1905, 9 2013.
16. L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for local queries. *ACM SIGMOD Record*, 15(2):84–95, 1986.
17. A. Madkour, W. G. Aref, and S. Basalamah. Knowledge cubes - A proposal for scalable and semantically-guided management of Big Data. In *BigData*, pages 1–7. IEEE, 10 2013.
18. T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. *ICDE*, pages 984–994, 2011.
19. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB*, pages 91–113, 2010.
20. N. Papailiou, Dimitrios Tsoumakos, P. Karras, and N. Koziris. Graph-Aware , Workload-Adaptive SPARQL Query Caching. *SIGMOD*, pages 1777–1792, 2015.
21. N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF + : High-performance Distributed Joins over Large-scale RDF Graphs. *BigData*, 2013.
22. P. Peng, L. Zou, L. Chen, and D. Zhao. Query Workload-based RDF Graph Fragmentation and Allocation. *EDBT*, pages 377–388, 2016.
23. T. Rabl and H.-A. Jacobsen. Query Centric Partitioning and Allocation for Partially Replicated Database Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*, 2017.
24. P. Ravindra and K. Anyanwu. Scaling Unbound-Property Queries on Big {RDF} Data Warehouses using MapReduce. In *EDBT*, pages 169–180, 2015.
25. L. Rietveld, R. Hoekstra, and S. Schlobach. Structural Properties as Proxy for Semantic Relevance in RDF Graph Sampling. *ISWC*, 8797:81–96, 2014.
26. K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework. *PSIEtA*, pages 1–5, 2010.
27. A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF Querying with SPARQL on Spark. *VLDB*, 9:804–815, 2016.
28. B. Wu, Y. Zhou, P. Yuan, L. Liu, and H. Jin. Scalable SPARQL querying using path partitioning. In *ICDE*, pages 795–806, 2015.
29. D. Yan, J. Cheng, M. T. Özsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng. Quegel: A General-Purpose Query-Centric Framework for Querying Big Graphs. *VLDB*, 2016.
30. M. Yang and G. Wu. Caching intermediate result of SPARQL queries. *WWW*, 2011.
31. K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *VLDB*, pages 265–276, 2 2013.