EFFICIENT QUERY PROCESSING OVER WEB-SCALE RDF DATA

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Amgad Madkour

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Prof. Walid G. Aref, Chair

Department of Computer Science

Prof. Sunil Prabhakar

Department of Computer Science

Prof. Tiark Rompf

Department of Computer Science

Prof. Sonia Fahmy

Department of Computer Science

Approved by:

Prof. Voicu Popescu

Head of the Departmental Graduate Program

To my ever supportive parents, Magdy and Fagr

To my beloved wife and best friend, Neveen

To my daughters, my heart and soul, Hana and Laila

ACKNOWLEDGMENTS

All praise is due to Allah, the most gracious, the most merciful. I thank Allah Almighty for giving me the strength, patience, and perseverance that allowed me to complete my degree.

I want to thank my father and mentor Magdy and my loving mother Fagr for their continuous prayers and support. I also want to thank my siblings Akram, Adham, and Yara for always being there for me. My family provided me with an unequivocal sense of security and support that came in all shapes and forms. They gave me the confidence to pursue my dream knowing that they will always have my back no matter what happens.

I want to thank my wife, and my best friend Neveen. She never complained about my long hours at work. Instead, she would encourage me to do more to achieve my target. She was always there for me providing words of encouragement and support during the most frustrating times of my degree. She accepted being away from every thing she loves just to support me. She sacrificed everything to see this dream of mine come true. I owe her everything iv achieved.

I want to thank my advisor, Prof. Walid Aref for believing in me and supporting me throughout this long journey. His door was always open whenever I needed guidance in my work or personal life. I owe him a debt of gratitude. He helped shape me to be the researcher I am today.

Finally, I would like to thank Randy Bond for allowing me to work on the development of the computer science portal that covered a large portion of my Ph.D. funding. He always put my needs first before the project requirements.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Madkour, Amgad Ph.D., Purdue University, December 2018. Efficient Query Processing Over Web-Scale RDF Data. Major Professor: Walid G. Aref.

The Semantic Web, or the Web of Data, promotes common data formats for representing structured data and their links over the web. RDF is the defacto standard for semantic data where it provides a flexible semi-structured model for describing concepts and relationships. RDF datasets consist of entries (*i.e.*, triples) that range from thousands to Billions. The astronomical growth of RDF data calls for scalable RDF management and query processing strategies. This dissertation addresses efficient query processing over web-scale RDF data. The first contribution is WORQ, an online, workload-driven, RDF query processing technique. Based on the query workload, reduced sets of intermediate results (or reductions, for short) that are common for specific join pattern(s) are computed in an online fashion. Also, we introduce an efficient solution for RDF queries with unbound properties. The second contribution is SPARTI, a scalable technique for computing the reductions offline. SPARTI utilizes a partitioning schema, termed SemVP, that enables efficient management of the reductions. SPARTI uses a budgeting mechanism with a cost model to determine the worthiness of partitioning. The third contribution is KC, an efficient RDF data management system for the cloud. KC uses generalized filtering that encompasses both exact and approximate set membership structures that are used for filtering irrelevant data. KC defines a set of common operations and introduces an efficient method for managing and constructing filters. The final contribution is semantic filtering where data can be reduced based on the spatial, temporal, or ontological aspects of a query. We present a set of encoding techniques and demonstrate how to use semantic filters to reduce irrelevant data in a distributed setting.

1. INTRODUCTION

The Semantic promotes common data formats for representing structured data and their links over the web. It consists of a set of components including the *Resource Description Framework (RDF)* and the *Ontology Web Language (OWL)* that both provide an ecosystem where applications can query and draw inferences from the data. RDF is an integral component in many areas, e.g., Semantic Search, *e.g.*, Google Knowledge Vault [1], Microsoft Satori, Smart governments (e.g., Data.gov [1]), and medical systems (e.g., BioDASH [2]). RDF datasets consist of entries (*i.e.*, triples) that range from thousands to billions. For example, the Linked Open Data cloud (LOD) community [2] provides more than 300 interlinked RDF datasets consisting of 32 billion triples. LOD covers a wide range of different topical domains including life sciences, geographic, linguistics, media, government, social networking, and user-generated content.

Distributed and cloud platforms, *e.g.*, Hadoop [3] and Amazon EC2 [4], provide a shared storage layer and advanced data processing components that help manage large-scale RDF data. However, these platforms introduce several performance challenges, *e.g.*, disk I/O and network shuffling overhead, and hence efficient RDF query processing and data partitioning techniques can significantly improve the query performance over these platforms.

RDF systems are classified into either centralized or cloud-based systems. Centralized RDF systems can provide efficient solutions for managing RDF data (*e.g.*, RDF-3x [3], TripleBit [4], gStore [5]). However, studies have shown that centralized systems cannot scale to web-scale datasets or answer complex RDF queries [6–9].

[1] `www.data.gov`
[2] `www.lod-cloud.net/state`
[3] `hadoop.apache.org`
[4] `aws.amazon.com/ec2`

In contrast, cloud-based systems encompass variously distributed storage backends and query processing architectures that are capable of operating over web-scale RDF data [10].

1.1 RDF Data Model

RDF provides a flexible semi-structured model for describing concepts and relationships. Data represented as RDF requires no schema to be defined in advance. Each RDF entry (*i.e.*, triple) consists of a subject, property (i.e., predicate), and an object. An RDF dataset is composed of RDF triples and can be described in multiple formats (*e.g.*, RDF/XML, N-Triples, Turtle, OWL). A set of triples form a directed graph where the property connects a subject to an object. An RDF dataset can also be perceived as a relational table with three columns (*i.e.*, subject, property, object). Table 1.1 illustrates an example RDF dataset.

Table 1.1.
RDF Dataset consisting of seven triples with two properties.

Subject	**Property**	**Object**
:John	:mention	:Mary
:John	:mention	:Mike
:Alex	:mention	:Mary
:John	:tweet	“T1”
:Mike	:tweet	“T2”
:Mike	:tweet	“T3”
:Alex	:tweet	“T4”

Each entry in an RDF dataset is either a resource (*e.g.*, `:Alex`) prefixed with `":"` for simplicity, or a constant literal (*e.g.*, `"T1"`). RDF data triples can be split into a set of relational tables. For example, the triples can be split by Vertically Partitioning (VP) the data [11]. VP partitions the RDF data triples by the property name. VP achieves good performance when compared to other RDF data-partitioning strategies, *e.g.*, *property tables* [12, 13] when using tables to store RDF triples. However, VP is

inefficient when the predicate is unbound (*i.e.*not specified) in a query pattern as all VP need to be scanned to find a match for that pattern.

SPARQL is a standardized query language for RDF that harnesses the simplicity of the RDF model to describe both simple and complex queries. A SPARQL query consists of one or more *Basic Graph Patterns (BGP)*. Each BGP consists of one or more triples that describe a query.

Example 1:

```
SELECT ?x ?y WHERE {
      ?x :mention :John .
      ?x :tweet ?y . }
```

Example 1 illustrates a query that consists of one BGP with two triple patterns. The query finds a set of people and tweets that include the resource `:John`. The `?x` and `?y` symbols are termed *unbounded variables*. SPARQL attempts to match the dataset entries (*i.e.*, graph) that satisfy the unbounded variables in a query. Unbounded variables are also used to indicate a *join* between two or more triple patterns. This join type is termed a *BGP join* as it exists within a BGP pattern. For example, Variable `?x` joins the first and second triple patterns in the BGP, indicating that a resource (*i.e.*, subject, property, or object) needs to match both triples to qualify.

BGP queries exhibit specific shapes based on the position of the unbounded variable in the triple (*i.e.*, subject, property, or object). The most common query patterns resemble shapes, e.g., a star (*i.e.*, subject-subject), linear path (*i.e.*, subject-object or object-subject), snowflake (*i.e.*, star-shaped with linear), or a more complex structure consisting of multiple shapes. The number of triples defines the diameter of a BGP joined consecutively. Joins that exhibit a *star-shaped* pattern have a diameter of one. On the other hand, *Linear paths* have a diameter equal to the number of triples included in its chain. BGP join patterns have a crucial performance impact on any RDF system. For this reason, some systems are biased towards specific query

shapes. The most common query shape is the star-shaped pattern. On the other hand, linear-shaped queries are more challenging as they involve a higher number of intermediate results compared to the different query shapes.

1.2 Research Contributions

This dissertation addresses the challenges associated with efficient query processing over web-scale RDF data. The research contributions of this dissertation are as follows:

- To support efficient online query processing of RDF data, we introduce *Wo*rkload-driven *R*DF *Q*uery Processing (WORQ, for short), a set of 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 specific join pattern(s) are computed. Furthermore, these reductions are not computed beforehand but are instead 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. Also, we introduce an efficient solution for RDF queries with unbound properties. 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.

- To support efficient query processing using offline-processed RDF data, we introduce *S*emantic *Parti*tioning (SPARTI, for short), a data management approach that is agnostic to the query shape. In SPARTI, the partitioning of the data evolves based on the join patterns found in the query-workload. Initially, SPARTI vertically partitions the RDF data, and then incrementally updates the partitioning according to the workload, which improves the query

performance of frequent join patterns. SPARTI introduces a novel partitioning schema, termed SemVP, that enables the system to read a reduced set of rows instead of an entire partition, and also skip entire data blocks. SPARTI proposes a budgeting mechanism with a cost model to determine the worthiness of partitioning. Using real and synthetic datasets, SPARTI is shown to be more efficient than the state-of-the-art system by up to an order of magnitude.

- To improve the scalability of RDF query processing over the cloud, we introduce KC, an RDF system that filters non-matching join entries as early as possible to reduce the communication and computation overhead. KC uses a filter-based approach to generate reduced sets of triples (or reductions, for short) to represent join pattern(s) of query workloads. KC can materialize the reductions on disk or in memory and reuses the reductions that share the same join pattern(s) to answer RDF queries. These reductions are not computed beforehand but are instead computed in an online fashion. Furthermore, KC can efficiently answer complex analytical queries that involve unbound properties. Based on a realization of KC on top of Spark, extensive experimentation using the WatDiv, LUBM, YAGO, and Bio2RDF benchmarks demonstrates an order of magnitude enhancement in terms of preprocessing, storage, and query performance compared to the state-of-the-art cloud-based solutions.

- To further reduce irrelevant data, we introduce distributed RDF query processing using semantic filters. RDF queries can include multiple aspects such as Spatial (e.g., Regions, Points), Temporal (*e.g.*, Dates), and Ontological (*e.g.*, RDFS). We create set membership structures that represent different aspects of RDF resources. A resource is added to a semantic filter if it contains triples that correspond to that aspect. Semantic filters utilize set membership structures to filter resources (subjects or objects) that do not match the corresponding aspect of the query. Furthermore, semantic filters can be deployed in a distributed setting to reduce irrelevant data locally at every machine.

1.3 Dissertation Outline

This dissertation is organized as follows. Chapter 2 introduces workload-driven RDF query processing. Chapter 3 introduces semantic partitioning of RDF data. Chapter 4 introduces KC RDF query processing system. Chapter 5 introduces semantic filtering. Finally, Chapter 6 presents concluding remarks for the dissertation.

2. WORQ: WORKLOAD-DRIVEN RDF QUERY PROCESSING

RDF datasets consist of entries (*i.e.*, triples) that range from thousands to Billions, e.g., the Linked Open Data cloud (LOD) community has more than 300 interlinked RDF datasets consisting of 32 Billion triples. LOD spans a wide range of topical domains including life sciences, geographic, linguistics, media, government, and social networking.

Processing RDF queries involve multiple scans of the same data, *e.g.*, when specific 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 to answer queries. Reducing the network shuffling overhead highly relies on how the data is partitioned across the nodes.

This chapter 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.

2.1 Problem Statement

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 [14]). 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 [15, 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. As-

suming that the data is vertically partitioned [10,11] (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 [1] for a Spark-based implementation of WORQ is publicly available for download. Our experimental setup includes two synthetic benchmarks, namely WatDiv [18] and LUBM [19], and a real dataset, namely YAGO2s [20,21]. 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 to partitioning and caching. The results illustrate how the presented optimizations provide at least an order of magnitude better results on the three aspects mentioned above when compared to the Hadoop-based state-of-the-art solution.

The contributions of this chapter can be summarized as follows:

- We present workload-driven partitioning of RDF triples that can join together 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 chapter proceeds as follows. Section 2.2 presents the related work. Section 2.3 presents the online reduction of RDF data. Section 2.4 presents workload-

[1] `http://github.com/purduedb/knowledgecubes`

driven partitioning in WORQ . Section 2.5 presents how WORQ answers unbound-property queries. Section 2.6 presents the experiments performed over the WatDiv, LUBM, and YAGO datasets. Finally, section 2.7 presents concluding remarks.

2.2 Related Work

Graph-based partitioning is an NP-complete problem [22], and hence hash partitioning heuristics [7,23] are employed instead of graph-based partitioning to partition RDF data efficiently. However, sophisticated partitioning techniques [24–27] cannot guarantee that no data will be shuffled when processing complex queries with multiple joins. Several techniques [28, 29] utilize the query workload to enhance the partitioning of RDF data. Also, one study [18] demonstrates the need to adapt to workloads to guarantee consistent performance continuously. Characteristic sets [30, 31] capture the set of properties that occur together for a given subject. Castilo *et al.* [32] perform evaluation using (offline) materialized results. 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+ [33] provides a result-based workload-aware RDF caching engine that manages to index frequent workload subgraphs in real time dynamically. However, caching the final results of RDF queries incurs significant storage overhead and cannot generalize to broader query workloads. Yang *et al.* [34] 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.* [35] 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.* [36] uses a non-relational algebra based on a TripleGroup data model to answer unbound-property queries.

2.3 Online Reduction of RDF Data

WORQ employs Bloom-join [15, 16] to compute the reductions between vertical partitions. Many cloud-based systems [10] use vertical partitioning (VP) [11] including the state-of-the-art [8]. 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 [8]). Bloom-join utilizes a probabilistic data structure, termed Bloom filter [16]. A Bloom filter does not 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 2.1 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 2.1 illustrates the entries 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

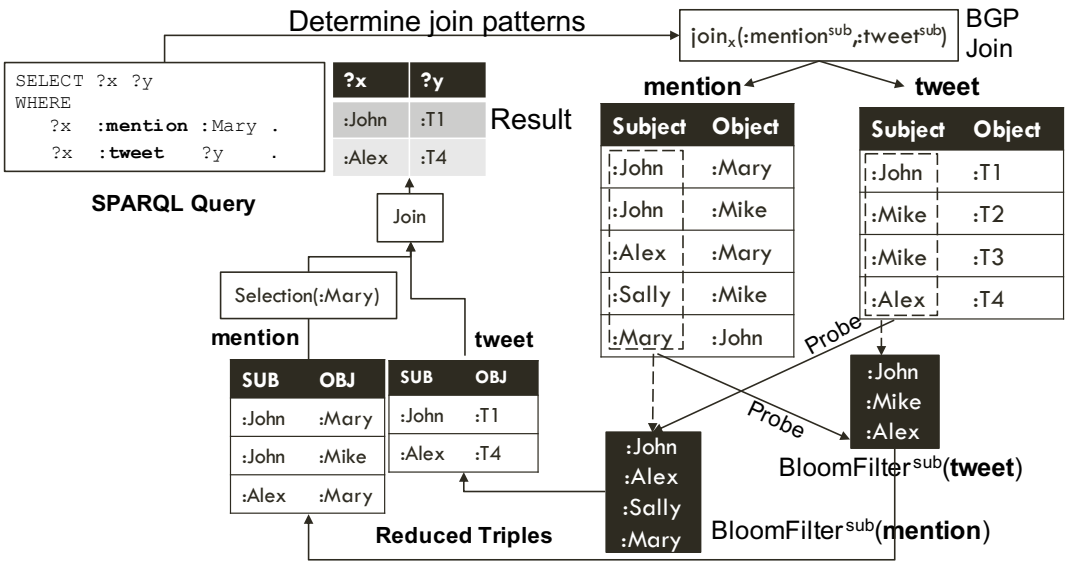

Fig. 2.1. Evaluating a SPARQL query using Bloom-join between :mention and :tweet.

vertical partition of `:tweet` is reduced from four entries to only two qualifying entries. The reductions for both properties are cached by WORQ 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 to 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 [37]. Figure 2.1 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.

2.3.1 N-ary Join Reductions

WORQ computes the reductions online instead of pre-computing the reductions offline [8]. Also, WORQ computes the reductions between all the possible (n-ary) query-triples instead of computing the reductions in binary form [8].

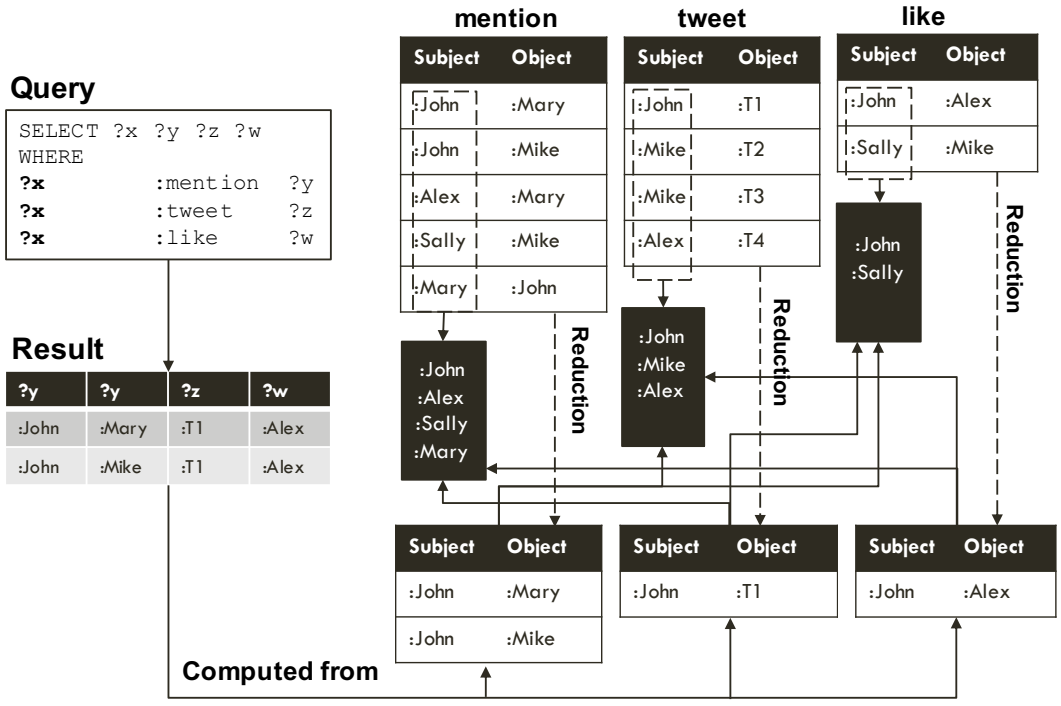

Fig. 2.2. N-ary join between the reductions of three query triples involving the :mention, :tweet, and :like VPs.

Figure 2.2 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 attribute and the `:like` Bloom filter on the subject attribute. 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.

2.3.2 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 when the query workload consists of frequent queries that do not necessarily share the same query pattern. WORQ is suitable for the former case where many different 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.

2.4 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.

Figure 2.3 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 2.3, 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.

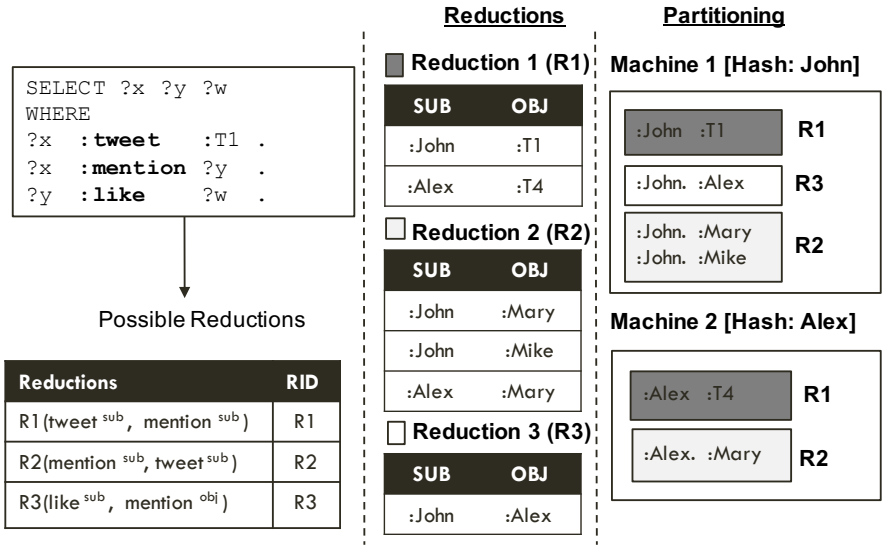

Fig. 2.3. Workflow for workload-driven partitioning.

2.5 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 [38], 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.

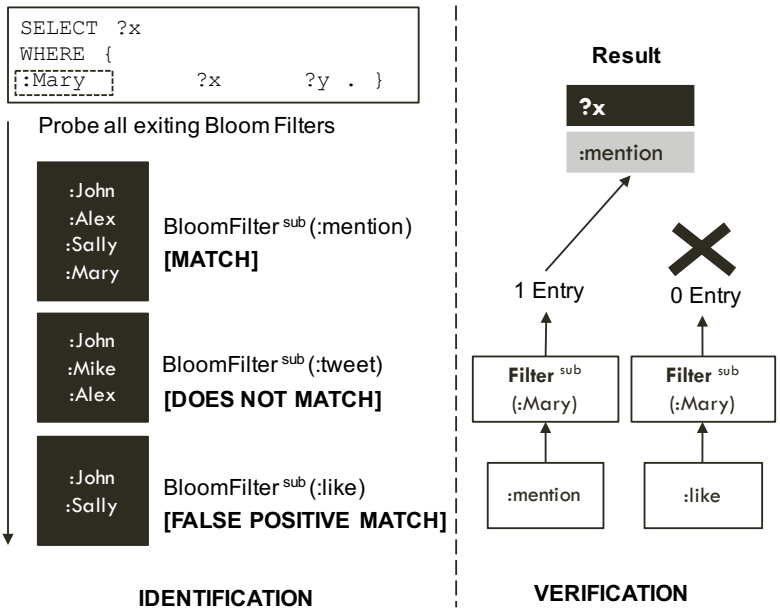

Fig. 2.4. The identification and verification steps to answer unbound-property queries.

Figure 2.4 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 2.4, `: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.

2.6 Experimental Evaluation

WORQ is compared against S2RDF [8], a Spark-based system that runs over Hadoop. S2RDF [8] proposes an extension to VP, namely ExtVP, where reductions of entries are computed for every vertical partition. S2RDF utilizes *semi-join reductions* [39] 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-Object). However, S2RDF exhibits a substantial preprocessing overhead. Semi-joins are expensive to compute and generate large network-traffic. Also, 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 [8]. Accordingly, this chapter 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 [2] columnar-store format. Vertical partitioning has been implemented as a baseline.

[2] `parquet.apache.org`

2.6.1 Experimental Setup

The experimental setup datasets and queries proposed by Abdelaziz *et al.* [9] are used. Our experiments are conducted using a real dataset (YAGO2s [20,21]) as well as two synthetic benchmarks (WatDiv [18] and LUBM [19]) 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 [18] 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 selectivities. For the unbound-property queries, we use the query workload provided by Alvarez-Garcia *et al.* [35] that represents 500 queries covering three combinations namely, an unbound subject with bound object, unbound object with a 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 [3, 8]. 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 to generate all possible query shapes.

Our experiments are conducted using an HP DL360G9 cluster with Intel Xeon E5-2660 realized over five nodes. The cluster uses Cloudera 5.9 consisting of Spark 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 three benchmarks is loaded into memory before execution.

2.6.2 Experimental Results

Preprocessing Performance

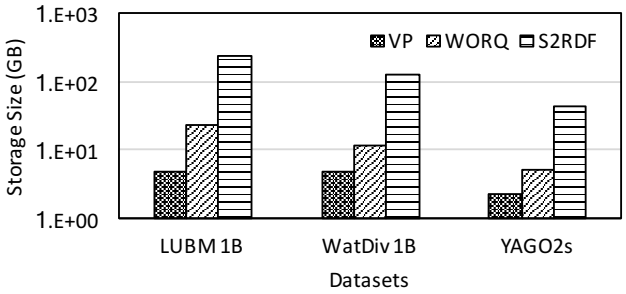

Fig. 2.5. Disk space utilization.

Figure 2.5 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 2.6 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

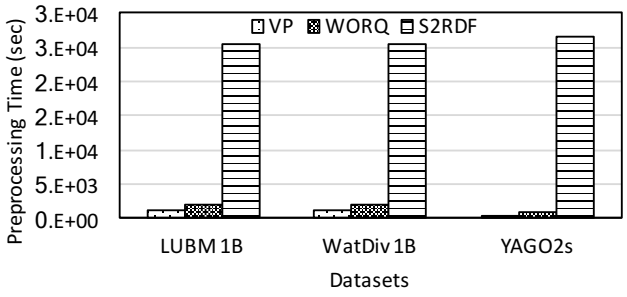

Fig. 2.6. Preprocessing time.

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 [8] 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.

Figure 2.7 and Figure 2.8 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.

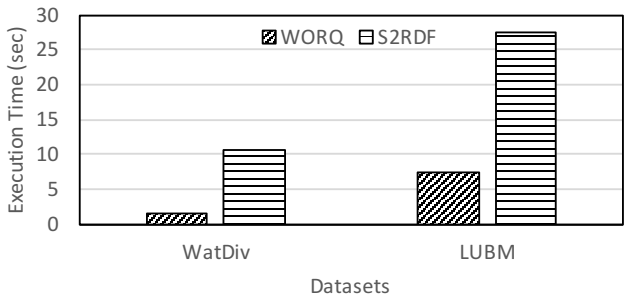

Fig. 2.7. Mean query execution time.

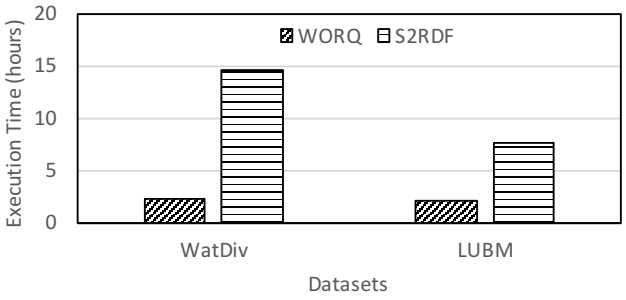

Fig. 2.8. Total query execution time.

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 [20, 21], the number of properties are in hundreds, making WORQ more appropriate to use than S2RDF.

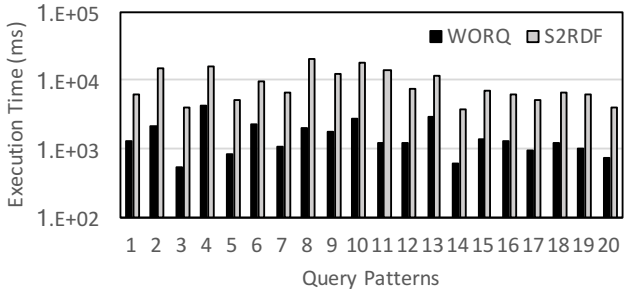

Fig. 2.9. Mean execution time per query pattern over WatDiv 1 Billion dataset.

Figure 2.9 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, 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 2.9 shows that WORQ executes each pattern nearly an order of magnitude faster than S2RDF.

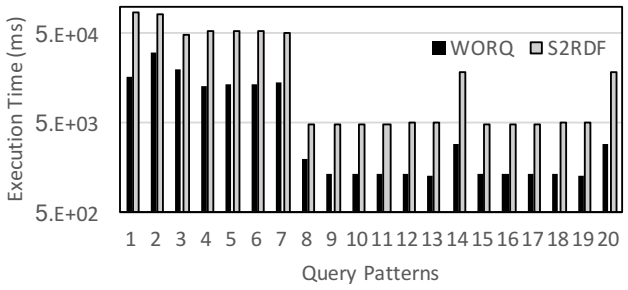

Fig. 2.10. Mean execution time per query pattern over LUBM 1 Billion dataset.

Figure 2.10 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 2.10 shows that all the patterns are executed faster by WORQ than S2RDF.

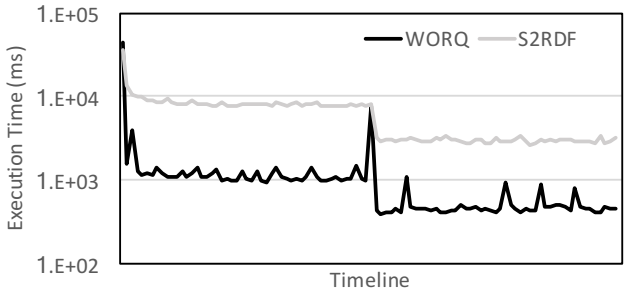

Fig. 2.11. Execution timeline for two query pattern over WatDiv 1 Billion dataset.

Figure 2.11 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 significant 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 a near-linear performance for the rest of the queries that share the same join pattern.

Figure 2.12 repeats the same experiment for two patterns over the LUBM benchmark. Similar to Figure 2.11, 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 2.8 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.

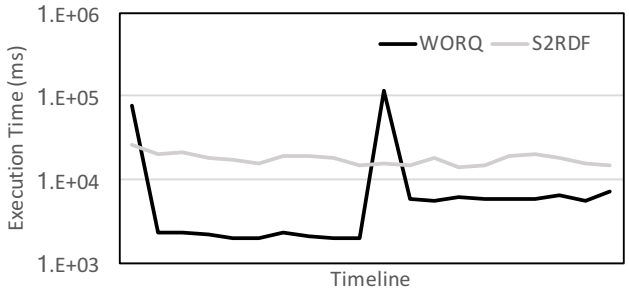

Fig. 2.12. Execution timeline for two query pattern over LUBM 1 Billion dataset.

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

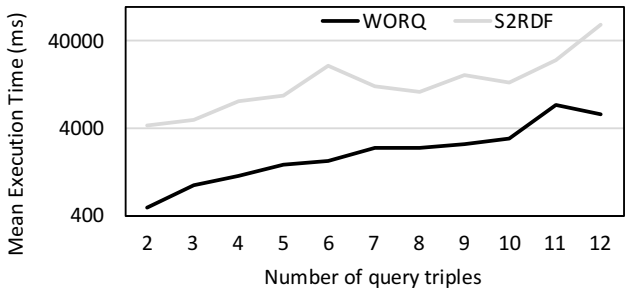

Fig. 2.13. Mean execution time - Number of triples per query over WatDiv 1 Billion.

Figure 2.13 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 2.14 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

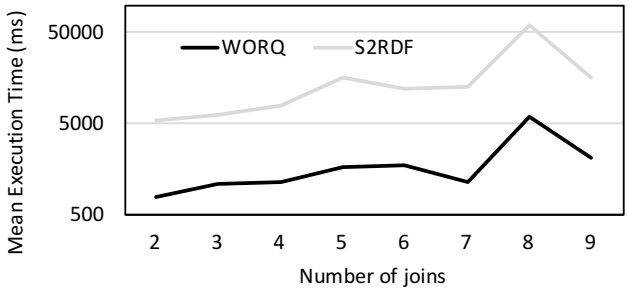

Fig. 2.14. Mean execution time for joins per pattern over WatDiv 1 Billion.

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.

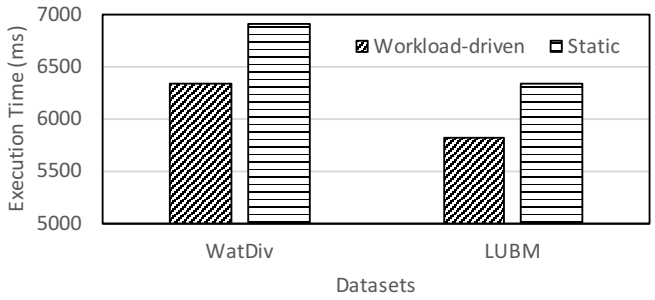

Fig. 2.15. Mean query execution time using workload-driven and static partitioning.

Figure 2.15 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 pre-specified criteria, *e.g.*, partitioning by subject. Static partitioning was performed on the subject

column. Figure 2.15 demonstrates that workload-driven partitioning contributes positively towards the overall query execution performance over the two datasets. The partitioning time is dependent on where data is stored initially on the cluster and generally incurs a minor cost. The query evaluation time given where data is partitioned dominates the execution time.

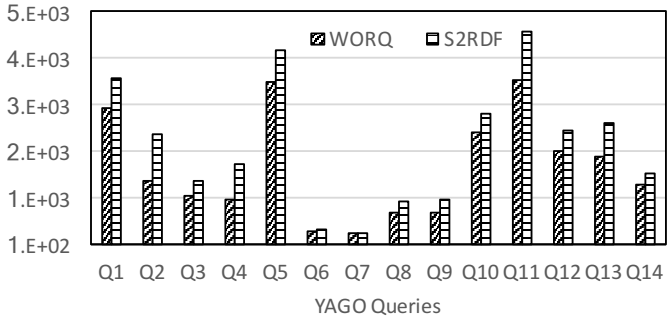

Fig. 2.16. Execution time of 14 query patterns over YAGO2s dataset.

Figure 2.16 gives a break-down of the query execution over 14 benchmark YAGO2s queries [8]. 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 five 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.

Caching of Reductions

Figure 2.17 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

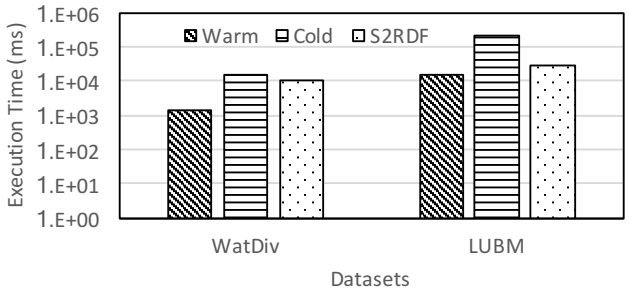

Fig. 2.17. Mean query execution time given warm and cold cache for WORQ over WatDiv and LUBM.

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 demonstrates how utilizing cached patterns (*i.e.*, reductions) achieves better query execution performance. The reason LUBM cold cache is worse is that 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 the price only once when a query pattern is seen for the first time. However, the cold-start cost is minor.

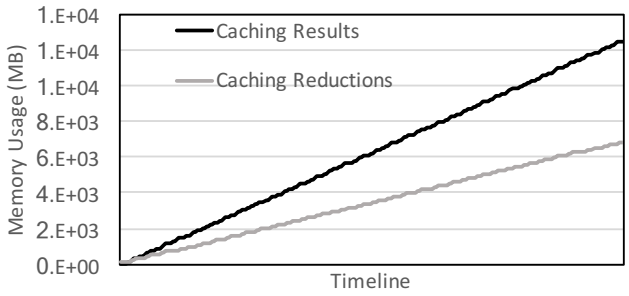

Fig. 2.18. Memory usage based on caching results and caching reductions.

Figure 2.18 gives a break-down of the memory usage over 5000 unique queries covering 100 patterns. Using a workload of 5000 different 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 many different queries share common patterns.

Performance of Unbound-Property Queries

Schatzle *et al.* [8] do not evaluate the performance of S2RDF for unbound-property queries as it is out of the scope of their current work. Also, 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 2.5 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 2.1.
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 2.1 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. The performance 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.*, the 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.

2.7 Concluding Remarks

This chapter presents several optimizations for RDF query processing over vertically partitioned triples. First, we show how to use Bloom join to compute reduced sets of (or reductions, for short) for specific 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 predefined partitioning criteria. Fourth, we show how to answer queries with unbound properties using Bloom filters efficiently. 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.

3. SPARTI: SCALABLE RDF DATA MANAGEMENT USING QUERY-CENTRIC SEMANTIC PARTITIONING

3.1 Problem Statement

Several techniques, e.g., Vertical Partitioning (VP) [11], split the RDF data by property name. VP achieves good performance when compared to other RDF data-partitioning strategies, e.g., *property tables* [12, 13]. However, due to the skewness in the data, where only a few properties constitute the majority of the data, VP introduces two coupled problems in a cloud-based setting. The first is that small partitions (i.e., properties that have few entries) are problematic for a distributed file-system, e.g., HDFS, where many small files will be generated, and a 64 MB block is allocated for every file/partition regardless of the number of entries in the partition. The second is that few partitions can contain the majority of the data and hence may cause significant disk I/O and network shuffling overhead when being processed. An important observation is that not all entries of a partition are part of the final result of a query. Also, only a small fraction of partitions are accessed in a real query-workload setting [14].

This chapter addresses three main issues, namely:

- How RDF data can be efficiently stored in the cloud ?
- How irrelevant data that does not contribute to the result of a query can be skipped ?
- How storage and query performance can be optimized based on the query workload ?

This chapter introduces Semantic Partitioning (SPARTI, for short), a platform-independent RDF system for the cloud. SPARTI includes three phases (1) *property-based partitioning*, (2) *partition reduction*, and (3) *query compilation*.

In the *property-based partitioning* phase, SPARTI employs a new relational partitioning schema, namely SemVP (Short for Semantic Vertical Partitioning). SPARTI partitions an RDF dataset based on the property name and stores the subject and object of a property name in a SemVP (i.e., every entry is composed of a *subject*, *property*, and *object*). SemVP extends each entry with row-level semantics, namely *join filters*. The join filters indicate whether an entry in a partition is part of a query join result or not.

In the *partition reduction* phase, the join filters are computed. First, SPARTI uses the query-workload to identify the related (*i.e.*co-occurring) properties for every RDF property. Then, a set of join filters are created between every property and all its related properties. The join filters are computed using Bloom-join [15]. The join result represents the reduced set of rows that qualify a join pattern and is materialized as a join filter. In the *query compilation* phase, SPARTI identifies the properties and join patterns in queries, and matches every property and query join pattern with a SemVP partition and a join filter, respectively. In case a match is found, a join filter (representing the reduced set of rows) is read to answer a query instead of reading the entire partition for a property. Also, using join filters allows skipping data at the block level when reading a property partition. This reduces the overall disk I/O when answering queries.

Example 1 gives an example where all people and their tweets that mention *John* are retrieved. The query consists of two entries (*i.e.*, triples) joined through a variable `?x`. SPARTI identifies `x={mention_S,tweet_S}` as a join pattern and creates a set of join filters for the properties included in the query, namely `:mention` and `:tweet`.

Example 1:

```
SELECT ?x ?y WHERE {
      ?x :mention :John .
      ?x :tweet ?y . }
```

```
Join filter (1): mention_S_JN_tweet_S
Join filter (2): tweet_S_JN_mention_S
```

where `S` denotes *subject*, `JN` denotes a join, and `?x` is a variable that joins both the `:mention` and `:tweet` entries. To illustrate, assume that SPARTI indicates that `:mention` and `:tweet` are related. Then, SPARTI computes the Bloom-join between `:mention` and `:tweet`, and creates a set of join filters representing the join pattern result. For example, SPARTI computes the Bloom-join for one of the filters, *e.g.*, $Joinfilter(1)$, between `:mention` and `:tweet` on the subject column. The join results for this filter are stored in a SemVP schema as part of the filters list of the `:mention` property.

```
SemVP(mention) = Filters[mention_S_JN_tweet_S]
SemVP(tweet)   = Filters[tweet_S_JN_mention_S]
```

The resulting join filters represent the reduced set of rows to read given a specific query join pattern. For example, the join pattern for `:mention` matches the name of $Filter(1)$ and the reduced set of rows is read instead of the entire `:mention` partition. The same applies to the `:tweet` property.

The contributions of this chapter are as follows: (1) SPARTI is introduced as a platform-independent RDF management system for the cloud, (2) SemVP is introduced as a new relational partitioning schema in SPARTI that provides row-level semantics for RDF data, (3) A new algorithm for discovering co-occurring properties in the query-workload is presented, (4) A cost-model is developed for managing join

filters, and (5) A comprehensive study is conducted that compares SPARTI with the state-of-the-art cloud-based RDF system using synthetic and real datasets.

Results demonstrate that SPARTI executes queries around half the time overall query shapes compared to the state-of-the-art while maintaining around an order of magnitude lower space-overhead.

The rest of this chapter proceeds as follows. Section 3.2 presents the related work. Section 3.3 presents the SemVP schema. Section 3.4 presents how related properties are identified. Section 3.5 presents how SPARTI evolves given a query-workload and defines a cost model for managing join filters. Section 3.6 presents the experimental setup and results. Finally, Section 3.7 contains concluding remarks.

3.2 Related Work

Cloud-based platforms introduce many challenges for RDF systems [10]. First, distributed file systems, *e.g.*, HDFS, do not provide fine-grained data access to file blocks, and thus requires a full scan of all blocks in order to read a file. Second, some of the blocks do not contribute to the query result. This generates two important issues (1) disk I/O overhead as all blocks are being read, and (2) network shuffling overhead when a join operation involving a table in a remote cluster node is required. Also, cloud-based RDF systems need to achieve a level of performance that is within the same order of magnitude as that of specialized systems in order to be useful.

There has been a recent surge of RDF stores over a variety of platforms. Recent surveys [40, 41] present various classifications of RDF-based systems ranging from relational [42], No-SQL [43] , to cloud-based systems [10]. RDF is accompanied by *SPARQL* [1], a standardized semantic query language that can express queries over RDF databases.

RDF stores can be broadly classified as distributed (*i.e.*, ones that involve multiple machines, e.g., a cluster) or centralized (*i.e.*, involves a single machine). The

[1] `w3.org/TR/rdf-sparql-query`

distributed approaches can be further classified by storage into (1) distributed file systems, (2) key-value stores, and (3) federated centralized stores.

Distributed File Systems: Distributed file systems (DFS) provide a scalable and reliable storage mechanism over commodity machines. A DFS is suitable for storing large files modeled as RDF. DFS systems, e.g., *Hadoop Distributed file system (HDFS)* splits large files into blocks and distributes them across the cluster nodes. The blocks are automatically replicated to achieve consistency and fault-tolerance. By default, the DFS does not provide fine-grained access to the data and hence has to read all blocks. Hadoop-based solutions that rely on indexing [44, 45] are not widely adopted as a tangible solution for fine-grained access to disk blocks. In the following sections, various methods for storing RDF data over DFSs are discussed.

A straightforward method is to store RDF data as one file, and retain it's native triple form. The DFS is responsible for splitting the file into a set of blocks and distributing the blocks among the cluster nodes. SHARD [38] runs over HDFS and extends how DFS handles triples by grouping the same subject onto one line. However, SHARD scans the entire file, possibly multiple times as it relies on the *MapReduce framework* [46], leading to degraded query performance. PigSPARQL [47] also relies on HDFS for storing RDF data. PigSPARQL utilizes two layers: (1) MapReduce for compiling the SPARQL queries, and (2) *Pig* [48] for optimizing the execution of the query. Rapid+ [49] extends Pig with nested triple group algebra to avoid the MapReduce iterative processing. However, both PigSPARQL and Rapid+ suffer from high query latency as they rely on the batch-oriented model of MapReduce.

Another popular method for storing RDF data is to split it into a more fine-granular set of files. RDF triples can be split based on the *property* (recall that a triple is composed of a *subject*, *property*, and *object*). In other words, each file represents a property, with the file name being the property name. The file content includes two columns representing the subject and object of the property. An alternative method is to split based on the subject or object. However, splitting horizontally on the subject or object can generate a large number of small files. Additionally, most SPARQL

queries specify the property rather than the subject or object. Unbounded property queries (*i.e.*, the property in a query triple is specified as a variable) are considered analytical queries and represent a small fraction in SPARQL query-workloads [11]. HadoopRDF [50,51] extends vertical partitioning by splitting based on the RDFS class (*e.g.*, *person*, *country*). All entries belonging to a specific class can be determined using the `:type` property. However, HadoopRDF join processing is performed in the reduce phase that incurs a high network-shuffling cost. HadoopRDF also suffers a query processing overhead due to the batch-oriented processing of MapReduce.

An emerging set of proposals suggests storing RDF data in a relational model over cloud-based platforms. These proposals include SQL systems, e.g., Hive, Impala [52], and Spark-SQL [53] for querying large-scale data. These proposals harness the extensive research conducted by the relational database community and can be used to help address semantic data challenges. Among these systems is DB2RDF [54] that proposes the creation of an entity-oriented schema from the RDF data. The entity-oriented schema stores objects that belong to the same property in the same column, thus optimizing the query execution performance over star-shaped queries. DB2RDF utilizes a direct secondary hash (DSH), for handling multi-valued properties. Also, DB2RDF can run over various back-ends, e.g., Spark, DB2, and PostreSQL. However, performance in DB2RDF is biased towards one query shape. Sempala [55] uses Impala to process SPARQL queries. Sempala stores RDF data as one large property table [12]. This allows star-shaped queries to be answered efficiently as answering queries requires no joins. The proposals mentioned above share one common pattern, namely having a predefined schema. S2RDF [8] proposes an extension to VP, namely ExtVP, where reduced sets of entries are computed for every vertical partition. S2RDF created reductions using *semi-joins* [39]. The reduced sets represent all possible join combinations that appear in SPARQL queries (*i.e.*, Subject-Subject, Subject-Object, Object-Object). First, S2RDF computes the semi-join reductions between all the pair of properties in a dataset. Then, S2RDF uses the reduced partitions to answer the SPARQL queries. S2RDF is not tied to a specific query shape.

S2RDF performance is compared against other cloud-based and centralized RDF systems where it achieves better query execution performance. However, S2RDF exhibits a substantial preprocessing overhead that is performed only once. Semi-joins are expensive to compute and generate large network-traffic. S2RDF generates a reduction for every possible vertical partition combination. On the other hand, SPARTI is similar to S2RDF with respect to employing reductions to achieve better query execution performance but without generating all possible combinations in advance.

Key-value stores: Key-value stores provide good performance for storage and retrieval of RDF data. In order to store RDF data, key-value stores create indexes that cover all the data and hence eliminate the need to store the data itself. Unlike centralized stores, key-value stores create fewer indexes. Hexastore [56] uses 3! indexes (denoting all RDF triple correlations) that cover all possible join combinations of a triple. This provides efficient data access and join performance for queries at the price of increased storage overhead, especially in a distributed setting. Most distributed RDF key-value stores utilize only three types of six indexes. Rya [57] utilizes Accumulo [2] and uses three clustered indexes to store the triples in the Row ID section of the key. Rya executes queries using an index nested-loops join at the server side. This generates a scalability issue because join processing is not distributed. H2RDF+ [58] utilizes HBase [3] and creates six clustered indexes to aggregate indexes for statistics. The statistics allow H2RDF+ to decide if a query should run in a local or distributed mode. H2RDF+ is best suited for selective queries while being able to run in a distributed fashion for non-selective queries. However, H2RDF+ distributed execution is much slower than its centralized counterpart.

Federated Centralized Stores: Federated stores utilize existing centralized stores that exhibit good query performance, e.g., RDF-3x [59], to execute in parallel. Federated stores follow a master/slave architecture, where the master node partitions the data while the slave nodes store and index the data in a centralized store. The

[2] `accumulo.apache.org`
[3] `hbase.apache.org`

challenge becomes how to efficiently partition the RDF data in a way that minimizes query execution time and network communication cost.

Federated stores highly rely on the efficient partitioning of RDF data. It becomes hard to determine a good partitioning scheme without knowledge of the query workload. As a result, many techniques utilize the *query-workload* to enhance the partitioning and replication, e.g., WARP [60], PARTOUT [61], and DREAM [62]. WARP [60] relies on efficient partitioning and replication techniques to minimize query execution time. WARP considers the query-workload to determine the triples that need to be replicated. PARTOUT [61] uses the query-workload to determine how to partition and replicate the data efficiently. Partitioning in PARTOUT includes two phases, namely fragmentation, and allocation. In the fragmentation phase, query-workload-based constants (*i.e.*, constant literal or URI) direct the horizontal partitioning of the data. In the fragment allocation phase, PARTOUT determines an allocation that maximizes locality when answering queries. The partitioning algorithm considers network and space constraints when assigning partitions to the available nodes. DREAM [62] partitions the queries instead of the data. DREAM relies on replicating all the data on every node and hence avoids shuffling intermediate results over the network. However, in a distributed setting, duplication of big data becomes an issue. DREAM uses RDF-3x as the centralized store and partitions SPARQL queries across nodes using a cost model.

3.3 SemVP

SemVP is a new relational partitioning schema that extends RDF vertical partitioning (VP) with row-level semantics. Similar to VP, SemVP partitions an RDF dataset based on the property name. For example, SPARTI generates two SemVP partitions for Table 1.1, namely `:mention` and `:tweet`. Each SemVP partition consists of two columns that represent the *subject* and *object* of an RDF triple while the *property* denotes the partition name. The SemVP schema is not tied to a specific

query shape. There are several advantages to property-based partitioning. First, it eliminates the $NULL$ issue introduced in property tables [12], where a subject may not have an entry corresponding to a property enclosed in the same table. Also, it simplifies query processing of SPARQL queries. The subject and object of a property are queried directly based on the property file name.

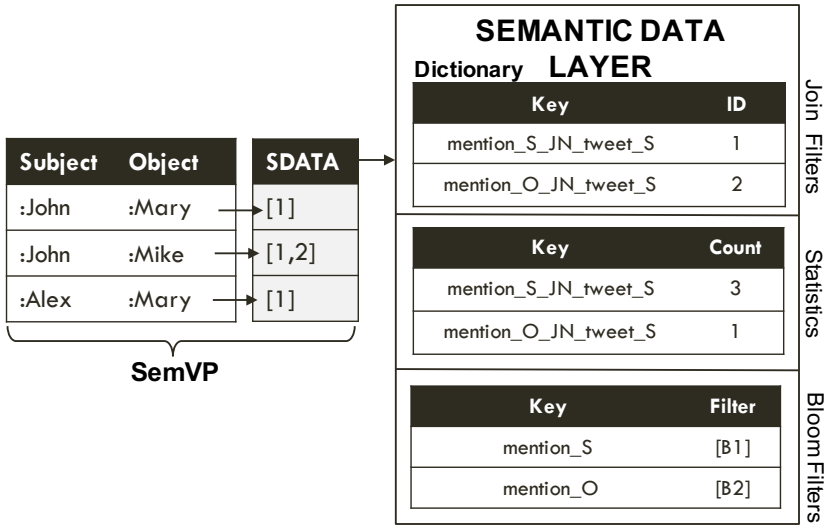

Fig. 3.1. SemVP Structure for :mention.

Figure 3.1 illustrates an example of a SemVP partition structure. SemVP extends RDF entries with a data structure, termed the *semantic data layer*, or $SDATA$, for short. SDATA is responsible for storing a set of row-level semantics, represented by *join filters*. Join filters allow SPARTI to determine the rows that contribute to the final result of specific query join patterns. Join filters are scalable, where reductions across n number of joined partitions (*i.e.*, properties) can be represented. Reductions represent a set of rows of a partition that satisfy a specific query join pattern. The value of n represents the maximum number of properties that appear alongside a particular property in the query-workload.

Regarding the physical storage, SemVP is best described using a column-store format [63]. Column stores are ideal for handling large-scale data due to their encoding and compression capabilities. The column store format provides efficient data encoding, where row repetitions of *subject* or *object* are recorded only once. For example, Table 1.1 introduces two entries for entity : *John* that get stored only once in a column store format.

3.3.1 Semantic Data Layer

The semantic data layer consists of three main sections. The first section is a dictionary representing the join filters. The join filters are defined as key-value pairs, where the key represents the join filter identifier, and the value represents whether a row qualifies a join pattern or not. Figure 3.1 lists a set of join filters. SDATA associates a join filter to a matching row in a partition. For example, the second join filter is associated with the second row only. In contrast, the first join filter is associated with all rows. The second section represents the statistics for every join filter. The statistics indicate the number of rows matching a join filter. SPARTI utilizes the statistics to determine the execution order of BGP triples. The statistics also allow SPARTI to decide the best filter to use given a specific query join pattern. The third section contains references to the Bloom filters created for the subject and object columns of a SemVP.

3.3.2 Join filters

Join filters consist of two components. The first component is the join filter *name* that represents a query join pattern. The second component is the association between a *row* and a *value*, where the row represents an entry in a partition and the value that indicates whether the entry matches the join pattern represented by the name. A join filter format is described as follows:

```
PROPERTY(1)_COLUMN-NAME_JN...PROPERTY(N)_COLUMN-NAME
```

The join filter name describes a join pattern between n number of properties where each property is joined by a specific column name (*i.e.*, subject or object). The convention defined by SPARTI is that the first property name included in the join filter name denotes the SemVP name.

For example, the join filter name `mention_S_JN_tweet_S` starts with `mention` as the property name followed by the join column name (*i.e.*, `S`). The remaining part of the join filter name represents the joined property name `:tweet` and the join column name (*i.e.*, `S`).

Join filters have several advantages. First, join filters can join n number of properties. This makes join filters highly scalable and able to provide reductions for long or short query join patterns. In contrast, a similar system, e.g., S2RDF, represents reductions between pairs of properties. Second, join filters are created between properties that do not need to be physically stored together in order to compute a join filter. Third, join filters evolve based on the query-workload, where filters are created, deleted without affecting the integrity of the partition data. Fourth, join filters are *elastic*, where it is possible to compute a subset of a join filter rather than computing all n properties. This allows SPARTI to consider smaller subsets of long join patterns observed in the query-workload to answer other smaller patterns.

Join Correlations

The BGP in a SPARQL query consists of triples that query subjects, properties, and objects. Joins between the subjects, properties, and objects represent the possible join correlations between triples. BGP allows defining a finite set of join correlations between BGP triple patterns. Given a pair of BGP-based triples, the possible join correlations for a subject and object are subject-subject (SS), subject-object (SO), object-subject(OS), and object-object (OO). It is possible to identify and precompute a set of BGP join correlations in advance [8]. Unbounded properties are considered

analytic queries and represent a small fraction of SPARQL query-workloads [11]. Similar to other work [8, 11, 64], SPARTI does not query unbounded properties.

SPARTI creates join filters using the BGP join correlations in the query-workload. For example, the join filter name `mention_S_JN_tweet_S` represents the join correlation for the query in Example 1 between `:mention` and `:tweet` on the variable `?x`. In contrast, systems, e.g., S2RDF, precompute all possible join correlations between all property pairs. Systems that use the RDF query-workload can provide good query performance [18]. Using the query-workload allows SPARTI to (1) identify the join patterns, and (2) limit the total number of join filters that need to be computed. Also, SPARTI creates join filters for both directions of the correlation. Computing both directions of the correlation has been proposed in distributed database systems [65]. It is also utilized in RDF systems, e.g., S2RDF. Consider the following example.

The set of reduced rows for the `:mention` property representing the join filter `mention_S_JN_tweet_S` is different from the set of reduced rows for `tweet_S_JN_mention_S`. The former represents the Bloom-join results for `:mention` with respect to the `:tweet` property while the later represents Bloom-join result for the `:tweet` property w.r.t. the `:mention` property.

Example 2:

```
SELECT ?x ?y WHERE {
      ?x :mention :John .
      ?x :tweet ?y .
      ?y :latitude ?a .
      ?y :longitude ?b . }
```

Example 2 gives an example SPARQL query with four triple patterns. The BGP joins are as follows:

```
x={mention_S, tweet_S}, y={tweet_O,latitude_S,longitude_S}
```

where `S` denotes a subject and `O` denotes an object. SPARTI generates a unique list of join filter names representing all permutations of every BGP join.

Note that both join filters are the same as they begin with the same property name and contain the same properties to join (in different order only). This is different from `tweet_S_JN_mention_S` and `mention_S_JN_tweet_S` as both start with different property names.

The maximum number of join filters for a combination of n properties is given by:

$$f(n) = n \times (2^{n-1} - 1) \tag{3.1}$$

The complete list of join filters for the query in Example 2 representing two BGP join combinations is as follows:

```
mention_S_JN_tweet_S
tweet_S_JN_mention_S
tweet_O_JN_latitude_S
tweet_O_JN_longitude_S
latitude_S_JN_tweet_O
latitude_S_JN_longitude_S
longitude_S_JN_tweet_O
longitude_S_JN_latitude_S
tweet_O_JN_latitude_S_JN_longitude_S
latitude_S_JN_longitude_S_JN_tweet_O
longitude_S_JN_latitude_S_JN_tweet_O
```

where the total number of join filters for the first combination (*i.e.*, `x` where n=2) is two and the total number for the second combination (*i.e.*, `y` where n=3) is nine with a total of 11 join filter names.

Computing Join Filters

SPARTI employs Bloom-join [15, 16] to compute reductions between partitions. Bloom-join determines if an attribute such as a subject or an object in one partition qualifies a join condition when attempting to join with another partition. Bloom-join utilizes a probabilistic data structure termed a Bloom filter [16]. A Bloom filter uses hash functions to store the input against n different hash functions. The main functionality of a Bloom filter is to check 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 percent of irrelevant rows that eventually are not joined in a Bloom-join.

SPARTI uses Bloom filters to probe the column entries of the query join patterns. SPARTI builds two Bloom filters corresponding to the two columns of a SemVP partition (*i.e.*, the subject and object columns) during the property-based partitioning phase. The Bloom filters are applied on the join columns attribute to filter the rows that do not qualify a join in both partitions. Furthermore, the results are materialized as join filters. Figure 3.2 illustrates an example of a Bloom-join and Bloom filters.

In Figure 3.2, the first step is to create a set of Bloom filters representing the subject and object of the SemVP partitions. SPARTI creates the Bloom filters during the property-based partitioning phase. In Figure 3.2, the Bloom filters: `mention_S`, `mention_O`,`tweet_S`, and `tweet_O` are created for the two SemVP partitions `mention` and `tweet`.

The second step is to identify the join filters. For now, assume that a relation between `:mention` and `:tweet` has been identified. The query consists of a BGP join between `:mention` and `:tweet` on the subject column. SPARTI utilizes the Bloom filter of `tweet_S` to compute a join reduction for the `:mention` property on the subject column (*i.e.*, `mention_S`). The `tweet_S` Bloom filter consists of three elements, namely `:John`, `:Mike`, and `:Alex`. Every element in the subject column of the `:mention` is checked against the `tweet_S` column.

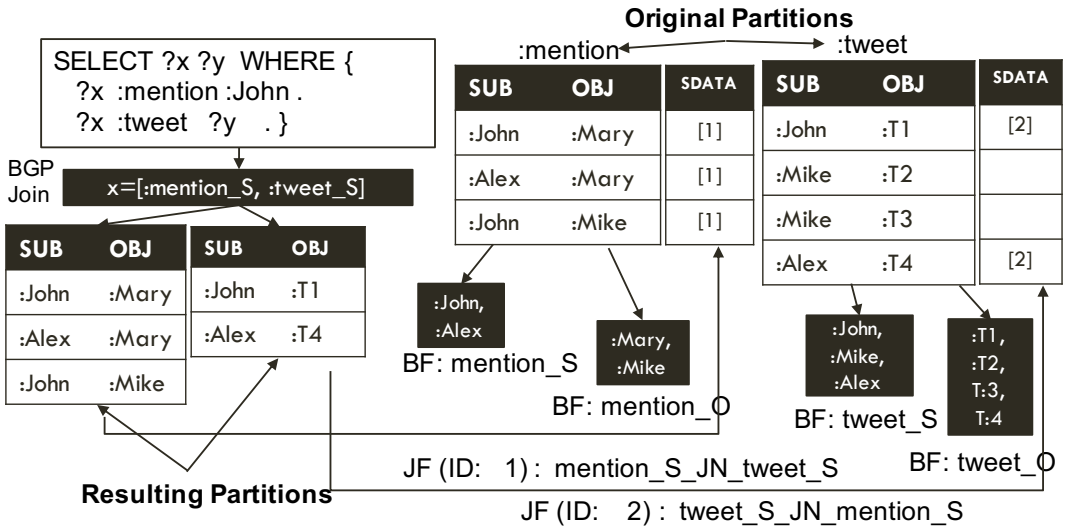

Fig. 3.2. Computing Bloom-join between :mention and :tweet - Bloom Filters (BF), Join Filters (JF).

The final reduced set for `:mention` indicates all the rows that qualify a join between the `:mention` and `:tweet` partitions. The reduced set is similar to the original partition, *i.e.*, there are no reductions possible. On the other hand, applying the same procedure in the opposite direction generates a reduced set for the `:tweet` partition, where the second and third rows of the original partition will not qualify a join between the `:mention` and `:tweet` partitions. The final reduced sets for both partitions are stored as join filters (marked as 1 and 2) and are saved in the semantic data layer (*i.e.*, the SDATA layer) of their corresponding SemVP.

3.3.3 Data Block Skipping

Several query engines, e.g., [52, 53] utilize hints provided by specific cloud-based file formats (*e.g.*, Parquet, ORC) over the stored data to speed-up query execution time. More recent work supports predicate-pushdown that allows *data skipping* at the *block* level [53, 66, 67]. A block is a horizontal partition that represents a set of entries

ranging from thousands to tens of thousands. Each block consists of meta-data, e.g., the *maximum/minimum* values of a block or a $NULL$ value if a block contains no entries in a specific column. This meta-data assists the query engine in determining if a data block can be skipped.

The SDATA representation in a SemVP allows a query engine to skip data blocks. Figure 3.3 illustrates how data block skipping is realized using the SemVP schema.

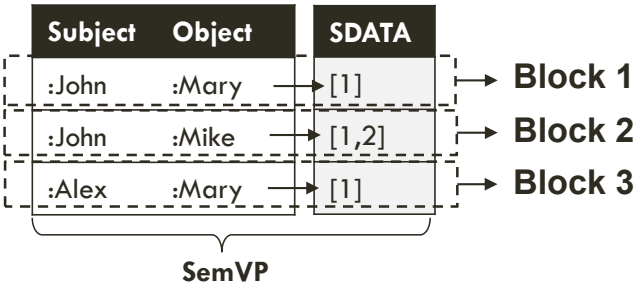

Fig. 3.3. Skipping blocks using SDATA.

For illustration, assume that every block consists of one entry. In Figure 3.3, a partition has three blocks. The query engine evaluates the meta-data of each block to determine the blocks to be skipped. For example, if a query pattern matches a join filter with ID 2, then the query engine skips scanning Blocks 1 and 3 as they do not contain the value 2.

3.4 Property Relatedness

SPARTI learns the relations among the properties from the query-workload. This section presents an algorithm for discovering then relations using BGP join patterns.

3.4.1 Co-occurrence Algorithm

SPARTI uses a *co-occurrence algorithm* that utilizes the BGP join patterns occurring in SPARQL queries to discover relationships among properties. A relation

between a property and all other properties is established based on the BGP join pattern. The co-occurrence algorithm does not require the query frequency (*i.e.*, support) to identify an important pattern. The advantage of not requiring the frequency is that *any* co-occurrence observed among properties is considered during the identification of the join filters.

Algorithm 1 lists the co-occurrence algorithm:

Algorithm 1 Identify co-occurring properties

Output: Related properties for every property

```
 1: function CoccurrenceApproach
 2:     Rel ← ∅                                          ▷ Empty proposal
 3:     Joins ← getQueryJoinPatterns()
 4:     for Join in Joins do
 5:         JoinProps ← getProperties(Join)
 6:         for Prop in JoinProps do
 7:             if Prop in Rel then
 8:                 Rel[Prop] ← Rel[Prop] ∪ JoinProps
 9:             else
10:                 Rel[Prop] ← JoinProps
11:             end if
12:         end for
13:     end forreturn Rel
14: end function
```

In Line 3, all the query-workload BGP join patterns are identified. A property is identified as co-occurring only if it is joined by a variable (*i.e.*, in the join condition) to another property. In Line 5, a list of all the identified properties in a BGP-join is created. Finally, for every BGP join, all properties that co-occur with a specific property in the `Rel` list are added as related (*i.e.*, co-occurring) properties.

3.5 Evolution Over Time

The query execution performance *evolves* when join filters are created for the frequent join patterns observed in the query-workload. Initially, SPARTI employs VP to answer queries using entire partitions. Then, SPARTI monitors queries in

order to identify the join patterns that do not invoke any join filter. Specifically, any property of a query that is part of a join pattern and does not invoke a join filter has its join pattern marked as a candidate for join filter creation. SPARTI initiates the partition reduction phase every k queries, where k is a user-specified parameter (*e.g.*, every 100 queries).

The rest of this section introduces a *cost-model* for *creating* and *deleting* join filters in a cloud-based setting.

3.5.1 Selecting Join Filters

Equation 3.1 illustrates the total number of join filters proposed for a given set of properties. There is a cost associated with creating and computing join filters, especially over cloud-based platforms. SPARTI provides a *budgeting mechanism* to decide on the maximum number of filters to compute. The cost and benefit of computing join filters are derived from the query-workload and the SemVP-based statistics. SPARTI adopts the following utility function to measure the importance of a join filter:

$$Utility = \alpha(S) + \beta(R) - \delta(P) \tag{3.2}$$

Equation 3.2 introduces three parameters, namely S, R, P [4]. Parameter S represents the support (*i.e.*, frequency) of a join pattern within a query-workload. S indicates the importance of a join pattern. Parameter R represents the partition size of the SemVP partition (*i.e.*, property) that the join filter belongs to (*i.e.*, the first property in the join filter name). Bigger partitions are more important to reduce than smaller ones. Parameter P represents the number of properties that the join filter has. A join filter is more valuable if it contains a smaller number of properties as it can be a sub-pattern for more join patterns. The objective of the utility function is to maximize the overall value of a join filter. The parameters α, β, and δ are smoothing parameters [68], where $\alpha + \beta + \delta = 1$ with $0 < \alpha, \beta, \delta < 1$.

[4]Parameters S, R, P are normalized between 0 and 1

A typical scenario in a cloud-based platform is to be constrained by a computational budget. SPARTI fits naturally in a cloud-based platform where the proposed cost model can be used to prioritize the computation of essential join filters. For example, the result of Equation 3.2 can be used to determine the importance of a join filter by adding its utility value to a *priority queue*. The join filter with the highest priority (*i.e.*, highest value) will be computed first. The number of join filters to compute is the size of the queue.

3.5.2 Eliminating Join Filters

Join filters incur a minimal storage overhead. Each join filter entry indicates whether a specific value of a join attribute has a matching value (joining tuple) in a partition or not. The match is represented by a 1 bit. However, as the query-workload patterns evolve, it becomes necessary to have criteria for deleting join filters. SPARTI uses Equation 3.3 to calculate when to delete a join filter.

$$T_{exp} = T_0 + [\delta(1 + (k-1)(1 - e^{-\lambda(Utility)})] \tag{3.3}$$

where T_{exp} is the expiration time of a join filter. T_0 is the most recent time-stamp a join pattern has been observed. δ is the default expiration time of a join filter (*e.g.*, one week). k is the maximum life-span of a join filter (*e.g.*, one month). λ is a smoothing parameter that sets the intensity of the cost (*e.g.*, the higher the lambda value, the longer the expiration time for low-cost filters). The intuition behind the equation is that the join filters that are expensive to create (taking into consideration the total number of scans and the support value of a join pattern) should have a longer time-span than the cheaper ones. In other words, the equation extends the expiration time of a join filter if it is expensive to compute.

3.6 Experimental Evaluation

SPARTI is compared to S2RDF [8]; a state-of-the-art cloud-based RDF system that runs over Spark. Previously, S2RDF has outperformed H2RDF+, Sempala, PigSPARQL, SHARD, and Virtuoso [8] where it has achieved on average the best query execution performance. S2RDF is built on top of Apache Spark [69]; an in-memory computational framework. It translates SPARQL queries to SQL and runs them on Spark SQL. To guarantee a fair setup, a Spark store and executor drivers are implemented for SPARTI. All Spark-related parameters are unified for both systems. Data is stored using Parquet [5] columnar-store format. We use a modified version of S2RDF query translator so that both systems would have the same query optimization algorithm. Vertical partitioning has been implemented as a baseline.

The semantic data layer (SDATA) can be implemented in multiple ways. One way is to create a column corresponding to every join filter of a SemVP partition. In each row, the value for a join filter column is set to `true` if the row qualifies a join filter or `null`, otherwise. This implementation is simple. Also, block-level skipping is achieved through *predict push-down* that is supported by most query systems. For example, a block is skipped if the meta-data indicates a `null` value for an entire block (*i.e.*, no rows in a block qualify a specific filter). Another implementation is to add SDATA as part of the meta-data of an existing file format (*e.g.*, Parquet). The SDATA layer can be realized similar to the $maximum/minimum/NULL$ meta-data provided by most query engines [52, 53]. Also, the query engine needs to be modified to understand the meta-data provided by the SDATA layer. SPARTI currently adopts the first implementation and utilizes the $NULL$ semantic for block-level skipping.

3.6.1 Experimental Setup

Experiments are conducted using four different datasets that represent three different sizes of a synthetic benchmark and a real dataset. The synthetic benchmark used

[5] `parquet.apache.org`

is WatDiv diversified stress testing [18]. WatDiv provides a benchmark dataset, test queries, and a stress-test query-workload. Three dataset sizes have been generated, namely 10 million, 100 million and 1 billion. For the query-workload experiments, the 10 million dataset is used. YAGO2s 2.5.3 [21] is the real dataset used. It contains 245 million triples. YAGO2s benchmark queries are used to compare the query execution time [8, 59].

An HP DL360G9 cluster with Intel Xeon E5-2660 was used, and five nodes were created. The cluster uses Cloudera 5.9 consisting of Spark 1.6 as a computational framework and Hadoop HDF as a distributed file-system. Each node consists of 32GB of RAM and 52 cores. The total HDFS size is one terabyte. All SPARTI phases are implemented in Java 8 as one package.

The experiments measure various aspects of SPARTI including (1) the number of generated rows, (2) the number of generated files, (3) the filesystem size, (4) the loading time, (5) the reduction size, (6) the caching time, (7) the workload performance, and (8) the query execution performance.

3.6.2 Experimental Results

We group the results based on three aspects: (1) *storage requirements*, (2) *query processing performance*, and (3) *query-workload*. For the synthetic dataset WatDiv, the query processing experiments are conducted on the largest WatDiv version, containing 1 billion entries.

Storage Requirements

Figure 3.4 gives the number of rows generated by the three systems (VP, S2RDF, and SPARTI). SPARTI does not introduce new rows, and thus maintains the same number of rows as those of VP. In contrast, S2RDF introduces new rows representing all the reductions that S2RDF computes.

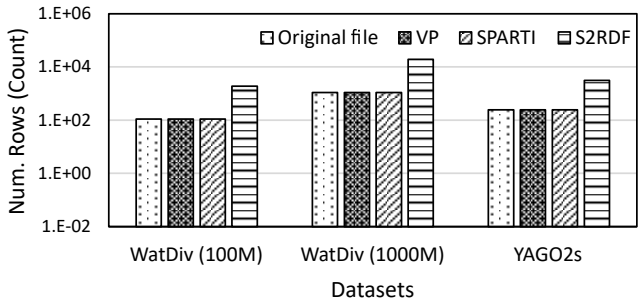

Fig. 3.4. Number of rows across all partitions.

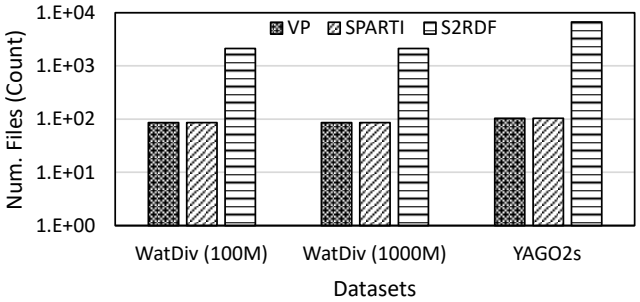

Fig. 3.5. Number of files across all partitions.

Figure 3.5 gives the number of files generated by the three systems. The original file is partitioned on the property name across all three systems. However, S2RDF stores the computed reductions in separate files, and thus stores more files than SPARTI and VP. The number of files can grow even further given a dataset with more properties, e.g., DBpedia [70].

Figure 3.6 gives the storage overhead incurred by the three systems. VP introduces minimal space overhead across all three datasets as it only needs to partition the original triple file based on the property name. SPARTI storage is composed of the original data, the bloom filters, and the join filters created per partition. S2RDF introduces the highest overhead due to the computed reductions in addition to the original data.

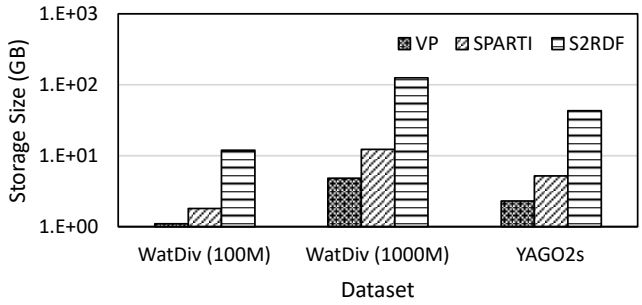

Fig. 3.6. Storage size.

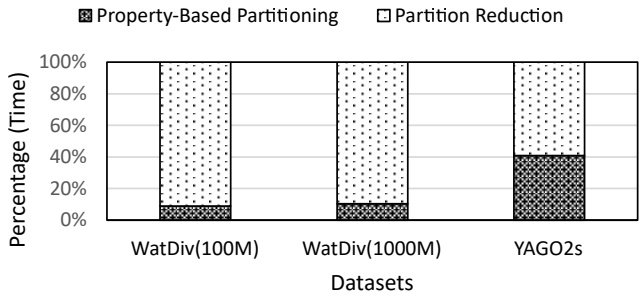

Fig. 3.7. SPARTI load time (in percentage) based on the operating phase.

Figure 3.7 gives SPARTI loading time based on the first two phases. SPARTI property-based partitioning constitutes 10% of the time over a well-structured dataset, e.g., WatDiv. In contrast, SPARTI property-based partitioning phase constitutes 40% over a less-structured dataset, e.g., YAGO2s.

Figure 3.8 gives the overall loading time for all three systems. The time taken by SPARTI includes both the property-based partitioning and partition reduction phases. VP has the smallest loading time due to its simplicity, followed by SPARTI, and then S2RDF. SPARTI's loading time for Dataset YAGO2s is almost an order of magnitude faster. This is attributed to the efficiency of Bloom-join reductions versus semi-join reductions employed by S2RDF.

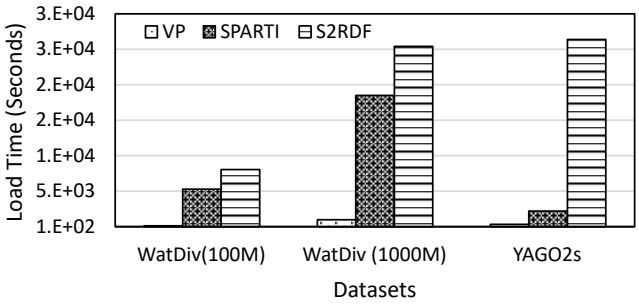

Fig. 3.8. Load time (Seconds).

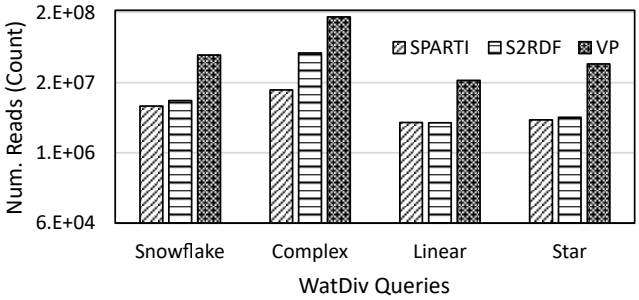

Fig. 3.9. Total number of entries read from WatDiv(1B).

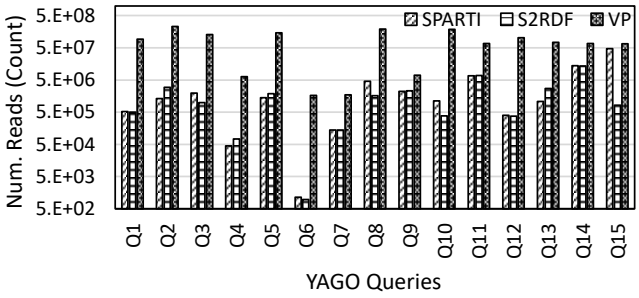

Fig. 3.10. Total number of entries read from YAGO2s.

Query Processing Performance

Figures 3.9 and 3.10 give the number of records read across the three systems. The results represent the average number of records exhibited across all queries of the

same shape. The reduction rates for both SPARTI and S2RDF are compared with VP (*i.e.*, representing the original partition).

In Figure 3.9, the reduction rates across all query shapes are better than that of S2RDF. The best reduction rate for SPARTI is exhibited with complex-shaped queries. This is attributed to two factors, namely the properties included in the query and the length of the join filters in terms of the number of properties included in the join filter.

In Figure 3.10, the reduction rates overall queries are better than those of S2RDF on average except for `Q15`. The performance of SPARTI with `Q15` is similar to that of VP. The reason is that one of the properties exhibited a high false-positive rate (*i.e.*, one or more repeating elements are wrongfully identified as qualifying for a join) that undermines the reduction effort over the entire query. In turn, this affected the query execution performance of `Q15` (see Figure 3.12). This result amplifies the impact of the reduction rate on the query execution performance. Decreasing the false-positive rate of the Bloom filters can mitigate this issue. However, space of the Bloom filter would increase. Thus, there is a trade-off between space and the rate of false-positives.

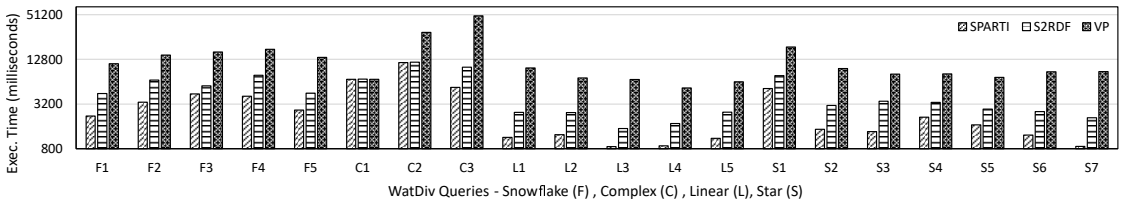

Fig. 3.11. Query Execution Performance (Milliseconds) over WatDiv(1B).

Figures 3.11 and 3.12 give the query execution performance of all three systems. Every query is executed five times and the average time is reported. In Figure 3.11, SPARTI executes queries around half the time of S2RDF over most WatDiv query shapes. The reason is due to the higher reduction rate that SPARTI exhibits when compared to S2RDF. Star-shaped queries, the most-common query pattern in RDF,

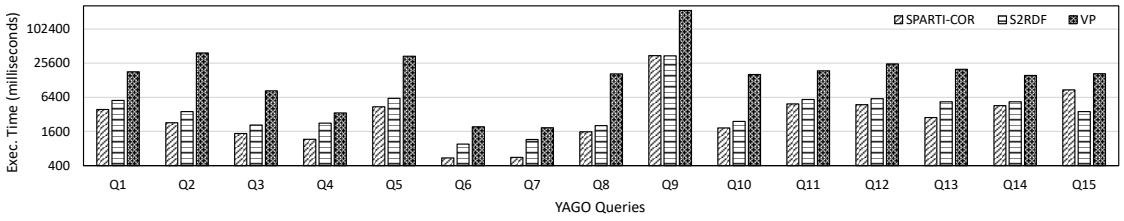

Fig. 3.12. Query Execution Performance (Milliseconds) over YAGO2s.

exhibits the highest query execution performance in SPARTI. This illustrates how star-shaped queries can benefit the most from input reductions, compared to other shapes. In contrast, complex-shaped queries exhibit competitive performance. This is attributed to the short join combinations. Both SPARTI and S2RDF provide reductions for short join combinations. SPARTI excels over queries with longer join patterns (*i.e.*, more properties) as they provide a higher reduction rate than that of S2RDF. Similarly, Figure 3.12 illustrates how SPARTI exhibits better query execution time than the other two systems with the exception of `Q15`. As mentioned earlier, the main issue is the low reduction rate that SPARTI exhibits when compared to S2RDF for `Q15`.

To illustrate the disk-block skipping capability over Spark, the query input partitions are cached entirely into memory and the total time for reading the entire partition is recorded. Spark SQL *predicate-pushdown* parameter is activated to take advantage of the block-level semantics of SPARTI. Similarly, S2RDF utilizes block-level skipping but only on the data level (*i.e.*, when reading a specific subject or object).

Figures 3.13 and 3.14 give the total amount of time needed to cache the partitions for the properties included in the queries. Both VP and S2RDF exhibit similar performance given the smaller sizes of their tables. SPARTI row-level semantics introduce an overhead when reading in the reductions as the overall size of a partition in SPARTI is larger than those of VP and S2RDF. However, using the block-level

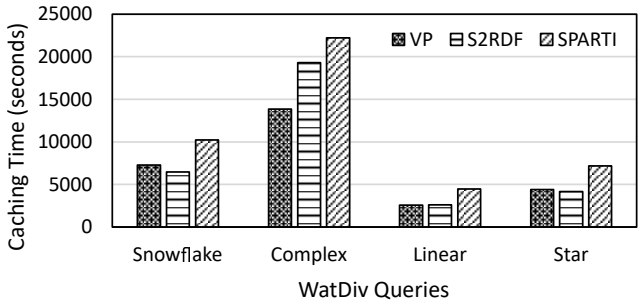

Fig. 3.13. Total caching time of records read from WatDiv(1B).

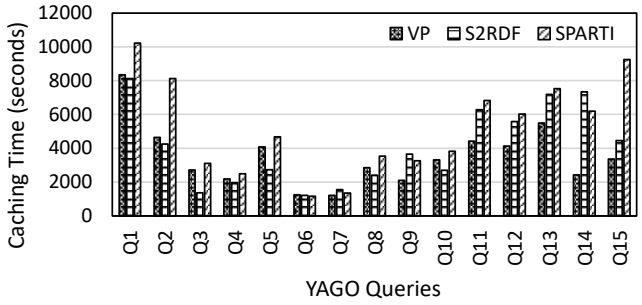

Fig. 3.14. Total caching time of records read from YAGO2s.

skipping semantics alleviates the overhead to an acceptable level across all query shapes.

Workload

The WatDiv benchmark provides a stress-testing workload [18] consisting of 125 structurally diverse query templates. The stress-test workload contains a total of 12400 queries that are divided into five parts (*i.e.*,`(P1-P5)`), where each part contains 2480 query. Each part represents an incremental query-workload. For example, `P2` consists of `P1+P2` while `P3` consists of `P1+P2+P3`. The WatDiv benchmark queries are executed and the average execution time across the queries of the same shape are computed.

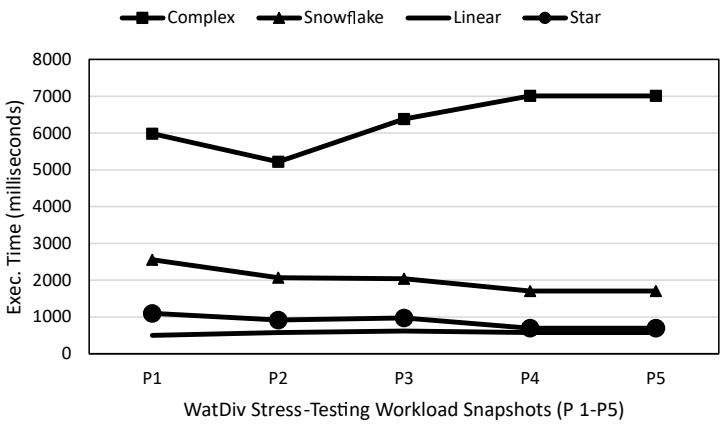

Fig. 3.15. SPARTI query execution performance using WatDiv stress-testing workload for different query shapes. P1-P5 are incremental snapshots of the workload.

Figure 3.15 gives the query execution performance for the WatDiv queries across five-time snapshots of the stress-test workload. The complex-shaped queries exhibit a temporary enhancement then an increase in the query execution performance. The reason is due to the same issue noted in Figure 3.10 where, for some properties, the rate of false-positives is higher for specific join patterns. In turn, this generates a temporary enhancement in point `P2`, then a degradation in performance between points `P3-P5`. As indicated earlier, one way to mitigate this issue is to decrease the false-positive rate of the Bloom filters. On the other hand, the star, linear, and snowflake shapes positively utilize the newly discovered patterns. These query shapes achieve an incremental query execution enhancement over time.

3.7 Conclusion

SPARTI is a platform-independent RDF management system for the cloud. SPARTI introduces SemVP as a new relational partitioning scheme that provides row-level semantics for RDF data. The row-level semantics, represented as join filters, provide SPARTI with a mechanism to read a reduced set of rows when answering specific query

join-patterns. An algorithm for discovering co-occurring properties in the query-workload has been introduced to compute reduced partitions between a set of related properties. The cost-model for managing join filters prioritizes the creation of the essential join filters to compute. Finally, the experimental study compares SPARTI against the state-of-the-art cloud-based RDF system where SPARTI achieves robust performance over synthetic and real datasets.

4. KC: EFFICIENT QUERY PROCESSING OVER WEB-SCALE RDF DATA

Join reductions can replace the original data in satisfying a wide range of RDF query shapes, *e.g.*, star, linear, and snowflake queries. Semi-joins can be used to precompute reductions between every pair of properties over all possible attribute combinations [8]. However, precomputing semi-join reductions incurs significant preprocessing time and storage overhead when employed over all the data [9]. Instead of precomputing all the possible reductions over all the data, KC uses set membership structures to compute the join reductions in an online fashion. Also, given a query, KC identifies the new join patterns (if any), *e.g.*, if `:mention` joins with `:like` on the object attribute, then a join reduction for `:mention` is identified when joining with `:like` on the object attribute, and another reduction for `:like` when joining with `:mention` on the subject attribute. Unlike [8], KC computes reductions between n-ary relations, called *n-ary joins*, instead of binary joins, resulting in smaller data sizes for each reduction given a specific join pattern.

4.1 Problem Statement

Generalized Filtering: The filtering approaches employed in various RDF systems can be classified into ones that use *exact structures* (*e.g.*, Bitsets [4], Bitmaps etc.) or *approximate structures* (*e.g.*, Bloom filters [71] etc.). However, no generalized filtering approach encompasses the *operations* (e.g., add, delete, contains, union) of both exact and approximate structures. System designers have then to choose between exact and approximate structures and integrate either structure across the whole system. A generalized interface that encompasses the common operations un-

der a unified filtering interface would be desirable because it can provide flexibility to choose the appropriate structure given the characteristics of the datasets.

Approximate set membership structures are used to determine the existence of an item within a set. For example, Bloom filters [16] do not physically store items, but instead hashes the input against different hash functions. Bloom filters are used to determine the existence of an item in a set. Bloom filters can have false-positives, but no false-negatives. Bloom filters are efficient in creation and probing time. Bloom join [15] is an example of how an approximate set membership structure can be used to reduce non-matching results during join evaluation.

Exact set membership structures can also be used to determine the existence of an item. For example, a Bitset structure can be used to determine the existence of an integer item within a set. Bitsets do not suffer from false positives and can accurately determine elements that exist in a set. However, Bitsets suffer from high storage overhead. In contrast, compressed bitmaps offer significant storage saving for sets. For example, Roaring bitmaps [72] provide superior compression performance when compared to an uncompressed bitmap while retaining fast lookup performance.

KC introduces *generalized filters* that can operate under both exact and approximate structures. For example, a generalized filter requires that the underlying structure (exact or approximate) support an add, contains, delete, and union operation. KC adopts a logarithmic approach for hierarchical construction of filters across cluster nodes instead of linearly collecting all filter elements on one node.

Generalized filters are constructed for the subjects and objects per property (i.e., predicate) of an RDF dataset. Each generalized filter represents the unique subjects or objects associated with an RDF property. The filters are broadcasted to all machines before answering any query. The broadcasted filters are used to push down the filtering of non-matching join entries on every machine in an online fashion thus reducing the overall computation and communication overhead of RDF queries.

Based on a realization of KC on top of Spark, our experiments have been performed using two synthetic benchmarks, namely WatDiv [18] and LUBM [19], and two real

datasets, namely YAGO2s [20,21], and Bio2RDF [73]. The purpose of the experiments is to study four performance aspects of KC : 1) the preprocessing time, 2) the storage overhead, 3) the effect of using exact and approximate set membership structures for reducing non-matching join entries, and 4) the query processing performance. The results illustrate how KC provides at least an order of magnitude better results in terms of preprocessing and storage and double the query execution performance when compared to the other Spark-based state-of-the-art solution [8].

The contributions of this chapter are as follows:

- We introduce KC, an RDF system for managing RDF data.
- We introduce generalized filtering that encompasses both exact and approximate set membership structures and define a set of common operations for both exact and approximate set membership structures.
- Based on evaluation over synthetic and real benchmarks, we experimentally demonstrate how KC achieves an order of magnitude enhancement in preprocessing performance and storage saving, and executes on average in less than half the time of the state-of-the-art distributed RDF systems.

Generalized filtering introduces a unified interface over set membership structures and a relational filter operator that can be implemented by RDF systems. Also, this chapter introduces a scalable approach for constructing filters in a distributed setting using the union operation. The experimental setup includes two real datasets instead of one. Also, all experiments were repeated over a larger cluster with more memory and processing power.

The rest of this chapter proceeds as follows. Section 4.2 presents an overview of KC . Section 4.3 presents generalized filtering. Section 4.4 presents the experiments performed over the WatDiv, LUBM, YAGO2s, and Bio2RDF benchmarks. Finally, section 4.5 presents concluding remarks.

4.2 Overview of KC

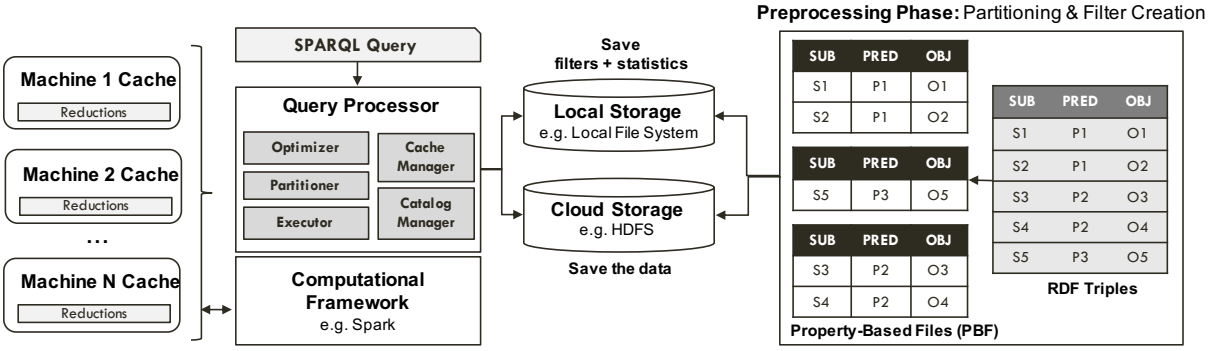

Fig. 4.1. An overview of KC .

Figure 4.1 gives an overview of KC . First, KC consists of a preprocessing step where the RDF triples are partitioned into smaller subsets. Second, KC consists of a query processor that identifies recurring join patterns in an online fashion.

The query processor contains a cache manager and catalog manager to optimize the execution. The cache manager is responsible for probing, adding, and removing reductions. The catalog manager is responsible for maintaining statistics about the PBS's and reductions in the system. The query processor utilizes the information provided by the catalog manager to infer join ordering (e.g., smallest first strategy).

4.2.1 Preprocessing

In the preprocessing step, the RDF data triples are split on the property (*i.e.*, predicate name) name. Unlike Vertical partitioning [11], the column name is retained.

For example in Figure 4.1, the RDF triples are split into three tables, one representing each predicate (*i.e.*, `P1, P2, P3`). A set of filters is also created and stored on disk to represent each column (except for the predicate column) of every property-based file (PBF). For example, two filters are created for the first table with predicate

P1. The first filter for the subject column contains elements S1, S2 and the second filter for the object column contains elements O1, O2. The filters store RDF resource names as they are used to check whether a particular resource exists during a join operation. The filters are used to compute a reduced set of rows for every PBF involved in the query evaluation.

When the tables are updated, the filter corresponding to a column that was updated will be recreated or updated, depending on the type of the filter (*i.e.*, exact set membership structure filter or approximate set membership structure filter). When a new table is added, then two filters will be created corresponding to the subject and the object columns of the new table. If a table is deleted, then both filters are deleted as well.

4.2.2 Query Processing

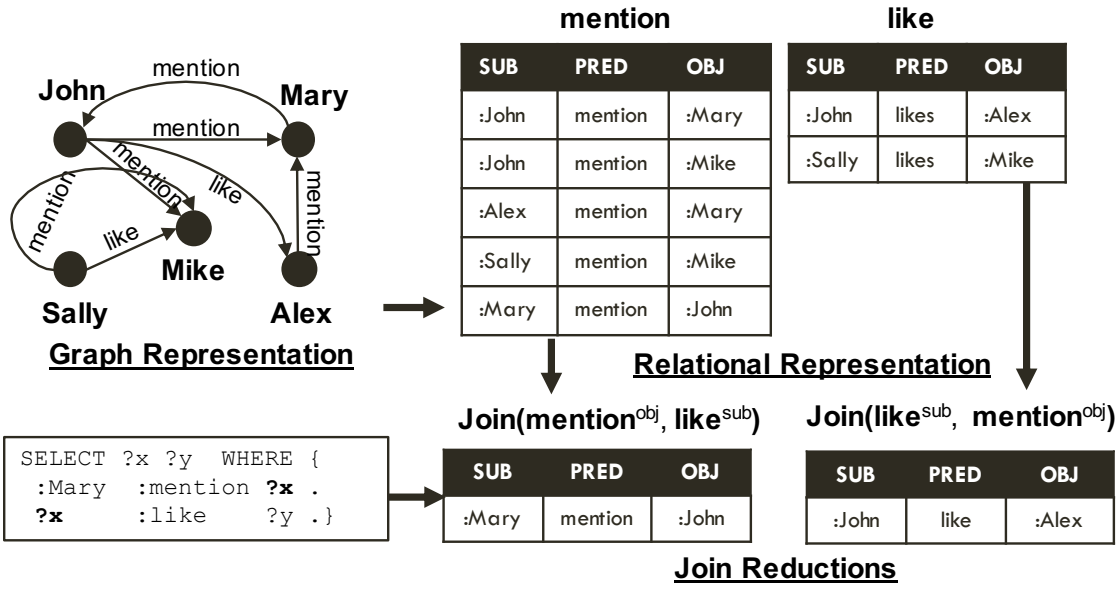

Fig. 4.2. KC adopts a relational approach for representing RDF graphs and uses reduced data triples instead of the original data triples to evaluate SPARQL queries.

KC processes SPARQL queries in an online fashion, where it requires no prior knowledge of the query workload. Given a query, the first step is to identify the attributes of the join patterns. In Figure 4.2, KC identifies `?x` as the join attribute between the query triples representing the `:mention` and `:like` properties. Next, the *cache manager* is probed to determine whether a specific join pattern reduction exists in the cache. The cache manager is used to return the reductions that match the join patterns. For example, the cache manager returns two reductions corresponding to the `:mention` and `:like` properties given that they are joined together. Then, the query executor uses the identified reductions to evaluate the query. If no reductions that match the join patterns exist, KC computes all the possible reductions for the identified join pattern. KC utilizes the filters during the join evaluation to eliminate the non-matching entries. In Figure 4.2, KC utilizes filters created for the `:like` PBF on the subject column to reduce the `:mention` PBF, and also uses the filter created for the `:mention` PBF on the object column to reduce the `:like` PBF. Once the reductions are created, KC caches these reductions in main memory so that they can be reused by other queries that share the same join patterns.

The KC query processor consists of a query *optimizer*, *partitioner*, and *executor*. The query optimizer determines the join ordering between the candidate tables (either property-based files or reductions). The query optimizer utilizes two types of statistics, namely *PBF statistics*, that are collected during the store creation process, and *reduction statistics* that represent reductions of PBFs given specific join patterns. The reduction statistics are used whenever a reduction of a join pattern is computed. The PBF statistics are used whenever a reduction does not exist for a specific join pattern. The query executor coordinates the query evaluation between all the query processing components. For every query triple, the query executor loads the corresponding data either from cache (through the cache manager) or from disk if no reductions exist for a specific join pattern. During the computation of the reductions, row counts representing statistics are collected. The counts are used by the query optimizer to determine an appropriate join ordering. KC employs a smallest-first strategy when

evaluating queries, where for every join pattern, the smallest reductions are joined together before the larger ones. Next, the query executor uses a *data partitioner* to partition (*i.e.*, split) the data triples of the reductions across the available nodes. KC adopts a query-workload-driven approach when partitioning data [74]. In particular, every PBF or reduction is partitioned based on the query join column. For example in Figure 4.2, data corresponding to `:mention` is partitioned based on object while `:like` is partitioned based on subject. Finally, the query executor passes the optimized SPARQL query to the computational framework, *e.g.*, Spark and the result is returned to the user.

4.3 Generalized Filtering

Generalized filtering encapsulates two classes of set membership structures, namely exact, *e.g.*, bitsets, and approximate, *e.g.*, Bloom filters, set membership structures. Generalized filters are used to remove non-matching entries from PBFs during join evaluation. Generalized filtering consists of two main components. The first component is a unified operations interface that is used to define the types of operations over the filters. Figure 4.3 illustrates the expected inputs and the four main operations proposed for generalized filtering.

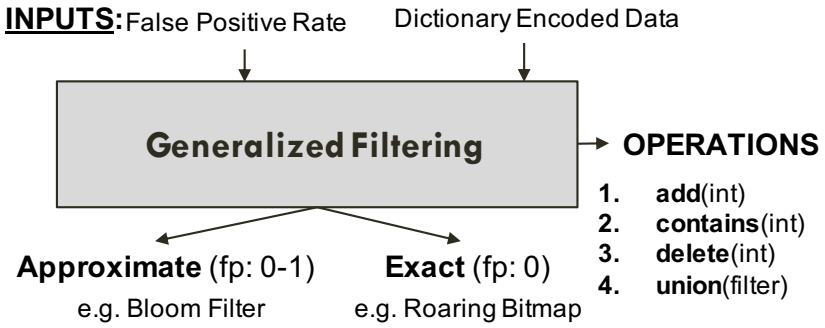

Fig. 4.3. KC Generalized filtering provides a unified interface for constructing filters.

First, generalized filtering operates over dictionary-encoded data (*i.e.*, data converted from string to integers). This is a standard operation performed by many RDF systems during preprocessing of RDF data. Second, KC uses the false positive (fp) input value when instantiating the approximate set membership filters. The false positive value is used to define the false positive rate of the created filter. If the false-positive value is zero, then exact set membership structures are used.

4.3.1 Operations

Generalized filtering proposes a set of operations that can be performed over the two set membership structures classes:

- **add(int)**: Add an element of type integer (returns: True if it succeeds/False if it fails)
- **delete(int)**: Delete an element of type integer (returns: True if it succeeds/-False if it fails)
- **contains(int)**: Check whether a specific integer is contained in the structure (returns: True/False)
- **union(filter)**: Concatenate two filters together (returns: concatenated Filter)

The union operation is used during in the construction of the filters. In a distributed setting, the typical scenario is to send all elements that will be added to the filter using an iterator approach where elements are sent to a central driver. However, KC uses a hierarchical approach when constructing filters where filters are merged on the machine level across cores before merging them across machines. Adopting a hierarchical approach reduces the overall size of elements being sent over the network when constructing filters. Also, using the union operation allows sending filters across the network instead of the actual elements where filters exhibit compression/hashing opportunities resulting in smaller network overhead, *e.g.*, when using roaring bitmaps or bloom filters.

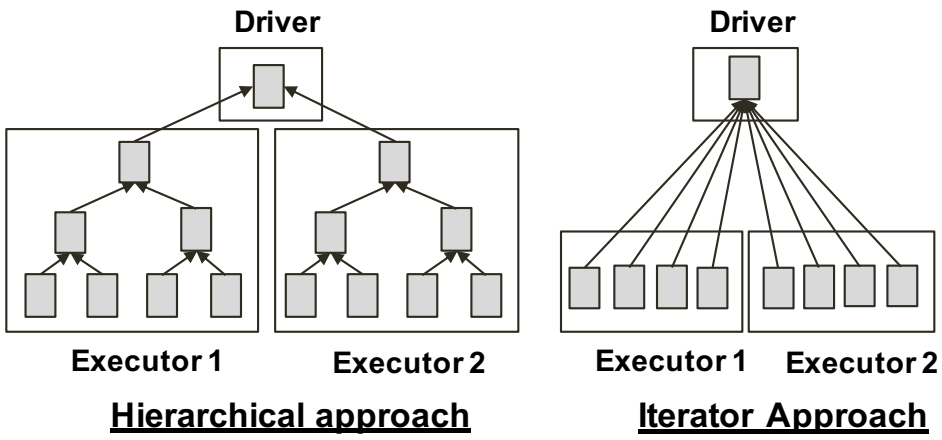

Fig. 4.4. KC uses a hierarchical approach instead of an iterator approach when constructing set membership structures in a distributed setting.

4.3.2 Filter Operator

The second component represents a relational filter operator (called `GenOp`) that is pushed down to the data sources (*i.e.*, PBF) to eliminate non-matching entries during join evaluation.

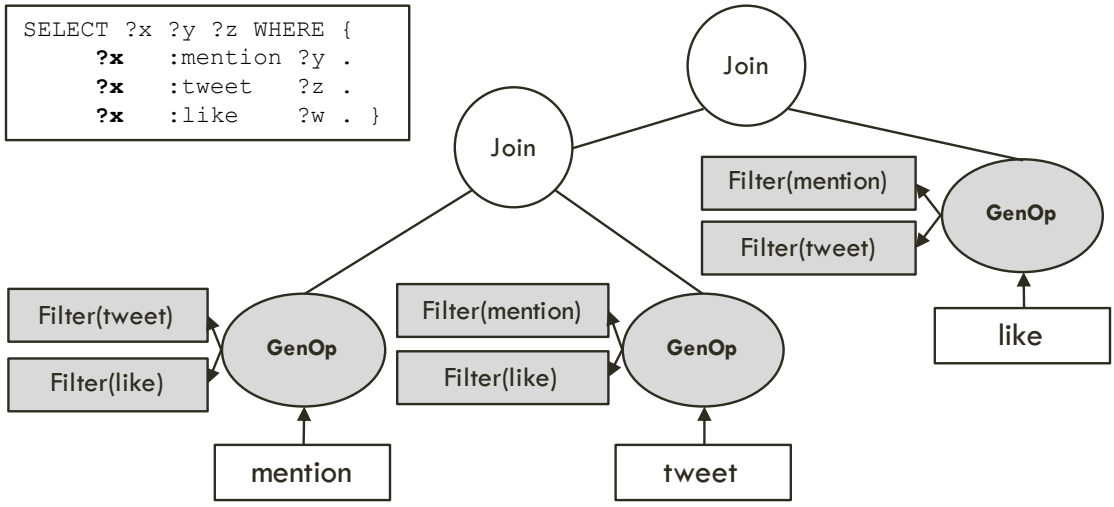

Fig. 4.5. GenOp is used by KC for filtering property-based files (PBF).

Figure 4.5 illustrates how `GenOp` is used for filtering PBFs in a relational query plan. The input to `GenOp` is a set of filters that correspond to the PBFs mentioned in the query. For example, the query includes three query triple patterns corresponding to three PBSs, namely `:mention`, `:tweet`, and `:like`. The first `GenOp` operates on the `:mention` PBF and reduces non-matching entries based on the filters corresponding to `:tweet` and `:like` PBFs. The output of `GenOp` is a *reduction* of `:mention` that represents the join between the `:mention`, `:tweet`, and `:like` on the subject column.

4.4 Experimental Evaluation

We realize KC using Spark [69]; a distributed in-memory framework that runs over Hadoop [75]. Hadoop-based systems are widely adopted both in academia and in industry. Also, realizing KC on top of a Hadoop-based system demonstrates the applicability of the proposed techniques over widely adopted components, e.g., Hadoop HDFS for storing the data, and Apache Spark as an in-memory computational framework. KC can also be implemented in other specialized RDF query processing systems that utilize the message passing interface (MPI), *e.g.*, Adpart [76], but implementing KC inside AdPart is beyond the scope of this work.

KC is compared against S2RDF [8], a Spark-based system that outperformed other Hadoop-based RDF systems.

4.4.1 Experimental Setup

Our experiments are conducted using two synthetic benchmarks that provide widely-adopted query workload generators and two real datasets:
1. **WatDiv** diversified stress testing [18]. WatDiv provides a stress-test query workload that allows generating several queries per-pattern.

The largest WatDiv benchmark consisting of one billion triple was used to demonstrate the query execution performance and preprocessing performance (i.e., the number of files generated, disk space utilization, and loading time). The workload pro-

vided by WatDiv [18] consists of 5000 queries that cover 100 SPARQL patterns where each pattern represents 50 variations. A variation represents different bound values for the same query pattern. The variations allow measuring the performance of specific patterns under different selectivity.

2. **LUBM** [19]. LUBM provides a query-workload generator, where 1000 queries are generated. A total of 1 billion triples were generated. Unlike WatDiv, LUBM does not specify the number of patterns.

3. **YAGO2s** YAGO2s 2.5.3 [20, 21] contains 245 million triples. YAGO2s benchmark queries are used to compare the query execution time [9].

4. **Bio2RDF** [73] is a biological database for interlinked life science data. A dataset consisting of 1 billion triples were used. Benchmark queries were used to compare the query execution time [9].

Our experiments are conducted using a cluster consisting of 12 nodes. The cluster uses Cloudera 5.9. We use Spark 2 as a computational framework and Hadoop HDFS as a distributed file-system. Each node consists of 168 GB of RAM and eight cores. The total HDFS size is 10 Terabyte.

4.4.2 Experimental Results

The experiments measure various aspects of KC and S2RDF including (1) storage overhead, (2) preprocessing time, (3) query execution performance. `KC` represents the baseline where PBFs are utilized. `KC-Roaring` indicates using `KC` with exact set membership structure filters such as roaring bitmaps. `KC-Bloom(0.01)` and `KC-Bloom(0.1)` indicate using `KC` with approximate set membership structure filters such as Bloom filter given a false positive probability of 0.1 and 0.001.

Preprocessing Performance

Figure 4.6 gives the disk storage overhead incurred by the four approaches over the LUBM, WatDiv, Bio2RDF, and YAGO2s datasets. PBF introduces minimal

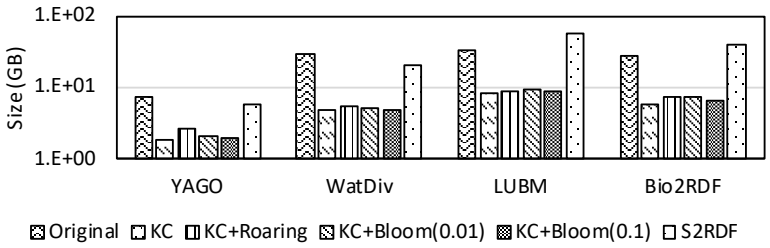

Fig. 4.6. Disk space utilization - (exact/approximate) setting, S2RDF, and the original data.

space overhead across all four systems. PBF partitions the original triple file based on the property name only. Storage using KC approaches (*i.e.*roaring and Bloom filters) is composed of the PBF and the filters. S2RDF introduces the highest disk storage overhead as it precomputes all the possible reductions and stores along with the original data.

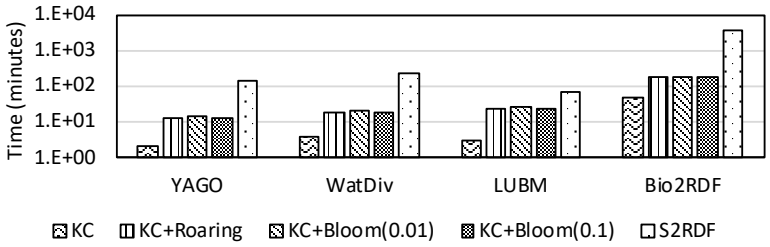

Fig. 4.7. Preprocessing time across the 4 benchmarks using KC (exact/approximate) setting, S2RDF, and the original data.

Figure 4.7 gives the preprocessing time over the YAGO, WatDiv, LUBM, and Bio2RDF datasets. S2RDF exhibits the highest preprocessing overhead as semi-joins need to be precomputed over all possible VP combinations. On the other hand, `KC` requires minimal preprocessing overhead as only the PBFs need to be created. `KC-Roaring` and `KC-Bloom` require additional computation to create the filters.

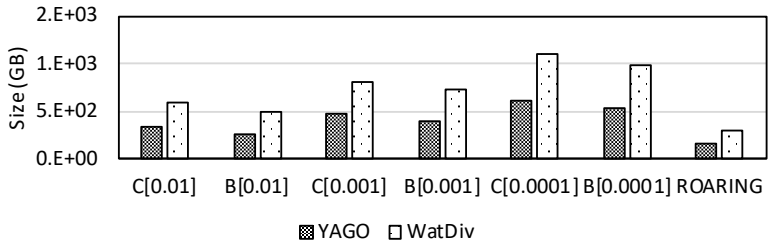

Fig. 4.8. Disk space usage for (1) approximate filters under different false-positive probabilities and (2) exact filter.

Figure 4.8 gives a size comparison between two approximate set membership structures, namely Cuckoo filters and Bloom filters. The comparison illustrates the size of the filters with respect to the false positive rate. Also, the size of an exact set membership structure, namely roaring bitmaps is presented. The results demonstrate how the false positive probability affects the size of the generated filters. In specific, the higher the false positive rate, the larger the filters sizes. The results for the approximate set membership structures demonstrates how Cuckoo filters exhibit a larger size than Bloom filters. In contrast, roaring bitmaps utilize compression that allows it to exhibit a smaller size than Bloom filters at a false positive rate of 0.01. The conclusion from this result is that exact membership structures can be utilized in situations where the false positive probability needs to be low (*i.e.*lower than 0.01) as it achieves low storage overhead. Otherwise, approximate set membership structures will outperform compressed set membership structures in terms of size. The size of the filter is an essential aspect as the filters are broadcasted to all machines before answering queries.

Query Execution Performance

The following experiments demonstrate the query performance of both KC and S2RDF across different benchmarks. S2RDF partitions every query during query

evaluation. We investigate the query execution performance under three experimental setups. The first is `KC-Filter` where filtering is performed in an online fashion when a new query join pattern is used. The first setup demonstrates the overhead generated by the filtering when answering specific query patterns for the first time. The second setup is `KC-Reduction` where reductions already exist for specific query join patterns and are directly used to answer the query. The third setup is `KC-NoStat` where no join ordering is enforced based on statistics. Instead, `KC-NoStat` ensures reordering is done to avoid cross products and reduce the number of joins, if possible.

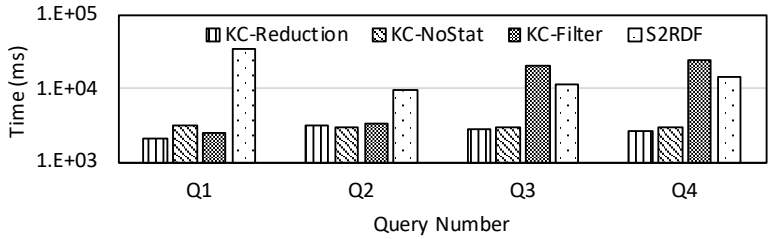

Fig. 4.9. Query processing performance (YAGO).

Figure 4.9 gives the total execution time over YAGO for 4 benchmark queries [9]. Query 1 consists of 5 unique patterns (*i.e.*, the predicate is not repeated) while Query 2 consists of 7 query patterns. On the other hand, Query 3 and 4 consists of 4 and five non-unique patterns respectively. `KC-Reduction` is consistently better across all queries. `KC-Filter` achieves better performance than S2RDF for Query 1 and Query 2 while it performs worse for Query 3 and Query 4. The reason is that the reduction sizes corresponding to the query join patterns for Query 1 and two are much smaller than those of S2RDF given the number of unique join patterns. The results between `KC-NoStat` and `KC-Reduction` is not significantly different as the original PBF and the reduction sizes do not generate a high query processing overhead.

Figure 4.10 gives the total execution time over Bio2RDF for five benchmark queries. The results show how the use of reductions allows achieving better query execution performance over a real and large dataset. `KC-Reduction` is consistently

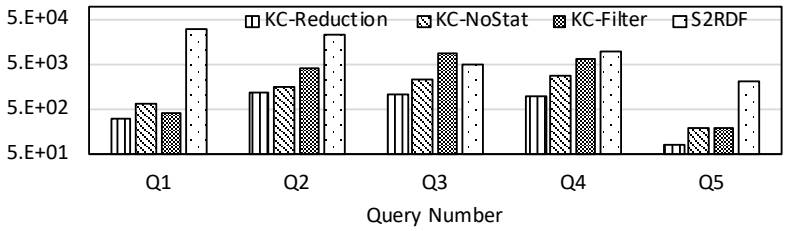

Fig. 4.10. Query processing performance (Bio2RDF).

better across all queries and `KC-Filter` achieves better performance than S2RDF for most queries. `KC-NoStat` generates a higher query processing overhead compared to `KC-Reduction`. This is attributed to not taking join ordering based on statistics into consideration leading to a worse query execution plan. The effect of `KC-NoStat` is more evident over Bio2RDF compared to YAGO due to the larger size of Bio2RDF.

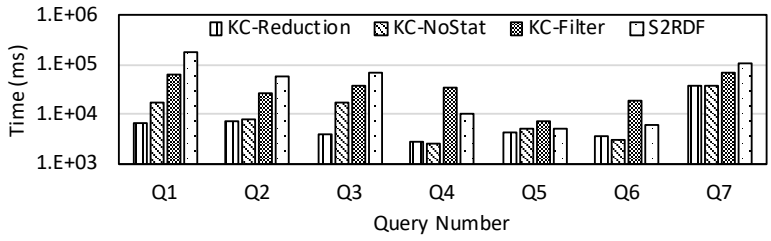

Fig. 4.11. Query processing performance (LUBM).

Figure 4.11 gives the total execution time over LUBM for 7 benchmark queries [9]. `KC-Reduction` is consistently better than S2RDF over the seven queries. Also, `KC-Reduction` is nearly an order of magnitude better than S2RDF for Query 1, 2, and 3. LUBM is larger than YAGO, so the advantage of utilizing reductions becomes more prominent, reaching nearly an order of magnitude better query execution performance at some instances. In Query 3, `KC-NoStat` generates double the overhead compared to `KC-Reduction`.

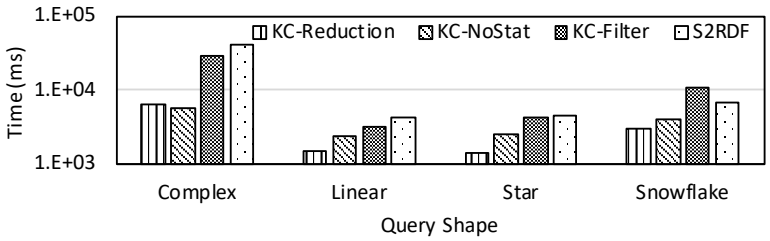

Fig. 4.12. Query processing performance (WatDiv).

Figure 4.12 gives the total execution time over WatDiv for 4 query shapes, namely complex, linear, star, and snowflake [9, 18]. For example, star-shaped queries have a common subject as a join attribute across a set of query triple patterns. For each shape, a set of 20 queries where used. The mean query execution time is reported for the 20 queries per shape. `KC-Reduction` consistently achieves the best performance across all query shapes. Also, `KC-Filter` achieves better performance across complex, linear, and star query shapes. `KC-NoStat` generates nearly double the query processing overhead compared to `KC-Reduction` for star-shaped queries.

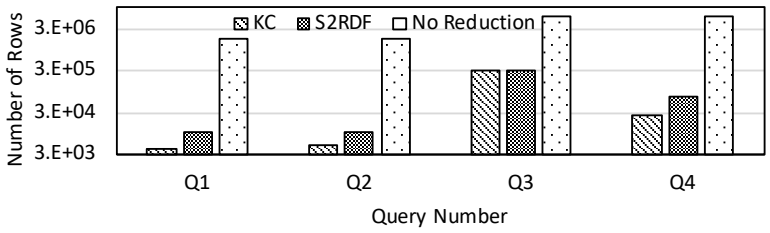

Fig. 4.13. Reduction rate (YAGO).

Figure 4.13 and Figure 4.14 give the reduction rate that both KC and S2RDF achieve compared to using no reductions. The reduction rate is calculated by counting the total number of rows involved in the join operation. The results demonstrate the main difference in the technique used by KC and S2RDF. In specific, KC employs a

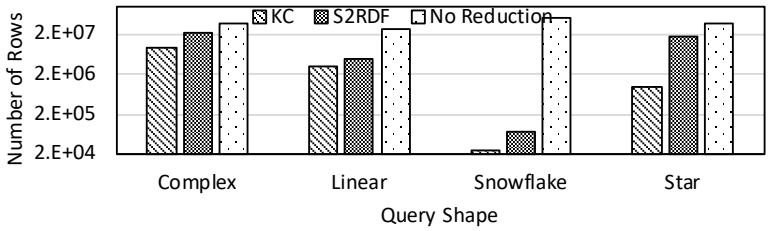

Fig. 4.14. Reduction rate (WatDiv).

more aggressive filtering strategy where N-ary joins are used to reduce every property based file involved in the query join pattern.

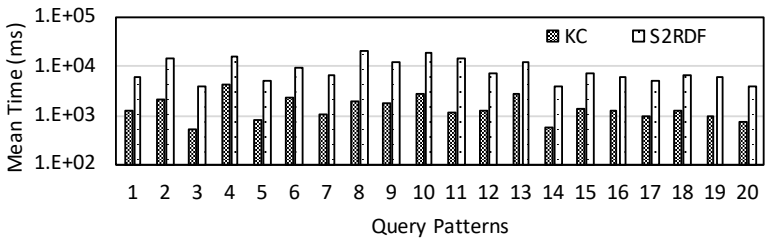

Fig. 4.15. Mean execution time per query pattern over WatDiv 1 Billion dataset.

Figure 4.15 illustrates the query execution performance over WatDiv (1 Billion triples) per query pattern. A query pattern represents a set of triples that vary based on the bound and unbound attributes. The mean execution time is recorded for the two systems for every triple pattern. Figure 4.15 shows that KC executes each pattern nearly an order of magnitude faster than S2RDF.

Figure 4.16 gives a break-down of executing 1000 queries over LUBM (1 Billion triples per query pattern). The mean execution time is also recorded on a pattern basis for the two systems. The number of patterns included in the LUBM query workload is 20. Figure 4.16 shows that all the patterns are executed faster by KC than S2RDF.

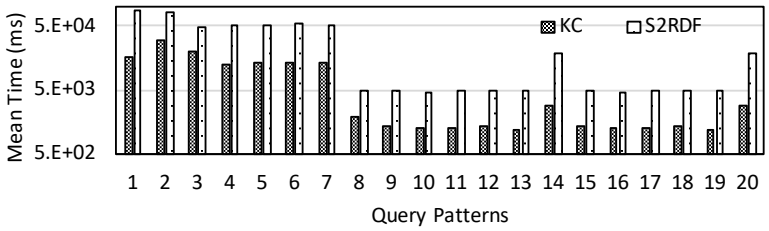

Fig. 4.16. Mean execution time per query pattern over LUBM 1 Billion dataset.

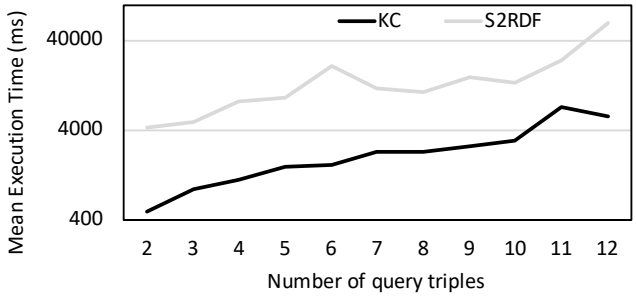

Fig. 4.17. Mean execution time based on the number of triples per query over WatDiv 1 Billion dataset.

4.5 Concluding Remarks

This chapter presented KC , an RDF query processing system for cloud-based platforms. First, we introduce the main components of KC . Second, we introduce generalized filtering where exact and approximate set membership structures are unified by defining a set of common operations. Third, we introduce a method for reducing non-matching join entries in an online fashion using the generalized filters. Extensive experimentation using the WatDiv, LUBM, Bio2RDF, and YAGO2s demonstrate how a realization of KC on top of Spark achieves an order of magnitude improvement in terms of preprocessing time, storage, and query performance compared to the state-of-the-art system.

5. DISTRIBUTED RDF QUERY PROCESSING USING SEMANTIC FILTERING

One of the main goals of semantic data is to allow computers to understand information with minimum human intervention. This includes providing informative answers to complex queries beyond what keyword-based queries can provide. One way of representing semantic data is RDF. Linked Data [1] allows RDF datasets to be interlinked to form an invaluable knowledge source. Projects such as the Linked Open Data Cloud (LOD)[2] provides access to more than a thousand web-scale RDF datasets spanning multiple domains including geography, government, life sciences, linguistics, media, and social networking. Businesses and Organizations (*e.g.*, Microsoft, Google, Wolfram Alpha) also utilize web-scale closed source semantic datasets for their business needs. Cloud-based ecosystems provide the necessary components for managing web-scale RDF datasets. The Hadoop Distributed Filesystem (HDFS) provides scalable, faulttolerant, and costefficient storage over commodity hardware. Computational frameworks (*e.g.*, MapReduce, Spark, Flink) provide efficient methods for analyzing and processing data. SQL-over-Hadoop systems (*e.g.*, Impala, Hive, Drill, Spark SQL) provide query capabilities over conceptually relational (*e.g.*, Spark Dataframe, Flink Distributed Dataframe) and graph data (*e.g.*, Graph-X).

A plethora of semantic datasets exist that can provide answers to complex queries beyond keyword-based results. However, the sheer size of semantic datasets raises scalability and performance challenges. Organizations are adopting the Resource Description Framework (RDF) for representing semantic data due to the simplicity of the RDF data model, and the expressiveness of the RDF query language. Existing cloud-based systems (*e.g.*, Hadoop) provide an efficient and scalable ecosystem for managing

[1] `http://linkeddata.org/`
[2] `lod-cloud.net`

RDF data. Also, distributed in-memory frameworks (*e.g.*, Spark, Ignite, and Impala) that run over Hadoop provide interactive performance for query-workloads. We introduce the concept of semantic filters that reduce the number of irrelevant entries that do not contribute to the final results of specific queries. The semantics associated with RDF data that does not match the semantics of specific RDF queries are skipped from processing. The reduction of the evaluated entries allows systems to process queries in around half the time.

In this chapter, we introduce how to efficiently filter data based on the semantic aspects of the query? These challenges include (1) encoding of a query's semantic aspects, (b) filtering based on the query's semantic aspects, and (c) distributed evaluation of the semantic aspects.

5.1 Problem Statement

RDF queries can include multiple aspects such as Spatial (e.g., Regions, Points), Temporal (*e.g.*, Dates), and Ontological (*e.g.*, RDF Schema (RDFS)). Utilizing the semantic aspects of both the query and the data provides opportunities to infer what resources can be identified as irrelevant with respect to the query. For example, if the query requires only resources that are within France, then it is safe to discard any other resource across all tables that are not within the spatial region of France. While this seems a straightforward and logical optimization, the challenge is in how to devise a generic approach that not only seamlessly applies to the spatial aspects but also the temporal and the ontological aspects.

Set membership structures can be utilized to store the encoded information rather than to store the encoding instead of the original data. However, the challenge becomes how to store the spatial, temporal, and ontological encoding inside a set membership structure. Also, how can set membership structures be used to filter resources in tables that do not possess any of three aspects.

Encoding: Each RDF triple can be represented using a string or a numeric representation (*i.e.*, dictionary encoding). However, the numeric representation of an RDF resource has no associated semantic meaning. Encoding mechanisms are used to associate semantics to numeric values that represent RDF resources. For example, a numeric value representing a Hilbert curve can be used to represent the spatial aspect of a particular resource [77]. However, the encoding needs to be applied to the original data rather than having a secondary representation for every resource. Another challenge is: How can more than one semantic aspect be associated with resources without modifying the original value of a resource?

Filtering: Encoding enables query processors to identify irrelevant data that do not match the query semantics. For example, the query in Figure 5.1 searches for Sculptures in a specific spatial region, *e.g.*, Egypt. Figure 5.1 illustrates the redundant data that do not contribute to the final result in three different tables. The challenge is: How to utilize the semantic aspects encoding, *e.g.*spatial information, to reduce redundant RDF triples when answering spatial queries in a distributed setting?

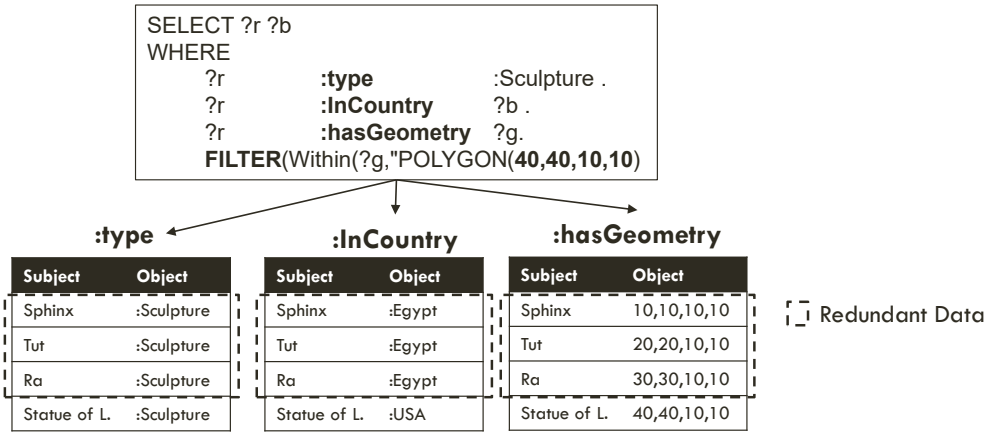

Fig. 5.1. Query processing of a SPARQL query with a spatial semantic aspect.

Similarly, the *temporal* aspect can be used to filter all Egyptian sculptures if the query is to find the sculptures created in the year 1875. The *ontological* aspect can

be used to filter the resources of the sculptures type, e.g., if the query is to find the resources that have Type Scientists.

Distributed Evaluation of the Semantic Aspects : Using a shared-nothing cloud-based approach to process queries is challenging due to the absence of indexes in many systems, *e.g.*, Hadoop. Systems that provide indexes still suffer from network shuffling overhead when locality of the indexed data is not enforced, especially if the index size is large. Another challenge is: How to efficiently perform specific operations, *e.g.*, spatial operations (spatial joins, knn), in a distributed fashion?

The contributions of this chapter can be summarized as follows:

- We present a set of encoding techniques for different semantic aspects including spatial, temporal, and ontological.
- We introduce semantic filtering of RDF resources that represent different aspects of the RDF data.
- We present an indexing mechanism for semantic filters for efficient RDF query processing.

The rest of this chapter proceeds as follows. Section 5.2 presents the related work. Section 5.3 presents semantic filters. Section 5.4 presents encoding techniques for semantic filters. Section 5.5 presents an indexing technique for semantic filters. Section 5.7 presents the experiments performed over the YAGO dataset. Finally, section 5.8 presents concluding remarks.

5.2 Related Work

Most systems [78–80] focus on providing semantic aspect support, *e.g.*, spatial, rather than focus on the performance aspect [77]. Parliment [80] implements the GeoSPARQL features in order to support spatial queries. Strabon [78, 81] is used to manage and query spatiotemporal data but proposes its own set of query constructs to achieve its target. Additionally, Strabon extends the query optimizer of existing

RDF toolkits such as Sesame [82] to supports the efficient execution of spatial predicates. However, both Strabon and Parliment rely on centralized stores, so they only support primitive query optimization. Broadt *et al.* [83] extends the RDF-3x store to support range queries. The system follows a spatial-first or non-spatial-first query evaluation method. Additionally, the query evaluation method does not tackle order-level optimization, *e.g.*interesting order, given the use of a spatial index. GeoStore [79] uses Hilbert space-filling curves for encoding geometry literals by the Hilbert order. S-Store [79] extends gStore [84] to support spatial querying.

Distributed spatial and spatiotemporal processing systems [85–88] have been proposed. However, they only focus on a limited set of semantics. Also, they are not tailored for handling RDF pattern matching queries that lead to query processing performance degradation.

Up to our knowledge, there does not exist a system that considers multiple semantic aspects of RDF data and queries to reduce irrelevant data during query processing. Also, there is no work on distributed processing of semantic aspects at scale.

5.3 Semantic Filters

In order to filter irrelevant data around specific semantic aspects, one approach is to create filters, termed semantic filters, around the semantic aspects of a query. Every semantic filter consists of resources that encompass a spatial, temporal, or an ontological aspect. The process of encoding maps the semantic aspect of a resource into a single dimension. The temporal aspect can be represented by a full timestamp (year, month, day, hour, minute, and seconds). Similarly, every ontological aspect can be represented using a single numeric value. Then, the semantic filters can be used to remove irrelevant resources from the query processing pipeline.

5.4 Encoding

The objective of encoding data is to translate a specific semantic aspect from higher dimensional form, *e.g.*, latitude, and longitude, into sortable one-dimensional data. The advantage of this approach is that it would be possible to encode specific query information and determine if it overlaps or matches the encoded data.

Once the code is generated, the points are split based on predefined criteria. One criteria for splitting the points is by count. This criteria has several advantages including simplicity and scalability. The scalability aspect relates to the set membership structure capacity. For example, using a probabilistic data structure such as Bloom filters puts restrictions on the size of the filter. In other words, the more elements a Bloom filter includes, the higher the false positive rate. The same applies to exact set membership structures such as Bitmaps where there can be instances where a maximum size is defined.

Finally, the encoded data is added to a set membership structure. The resource name is added as an element, and the range of encoded values is recorded. Adding the resource name allows us to determine if a specific resource is contained within a specific encoding range. This approach allows using the filter as an indicator of whether specific resources exist in a specific query range or not.

5.4.1 Spatial Encoding

Properties such as `hasGeomety` represent triples describing the latitude and longitude of RDF resources. The latitude and longitude of every RDF resource can be mapped to a single dimension using a Hilbert space-filling-curve (HSFC) [77] or Geohashing. For example, HSFC allows generating an ordered code that represents the spatial aspect of an RDF resource. Geohashing uses a short alphanumeric string to represent spatial coordinates. Such techniques can be utilized to determine if an RDF resource qualifies a spatial filtering operation specified in a query (*e.g.*, WITHIN() filter function). This is possible by verifying if the coordinates of a spatial RDF

resource falls within the boundaries of the query coordinates encoded values. The encoded data can determine if specific resources could be skipped during query evaluation. For example, if the HSFC encoding of a query range is between 15 and 19, then any entry below 15 or above 19 can be safely discarded from the computation.

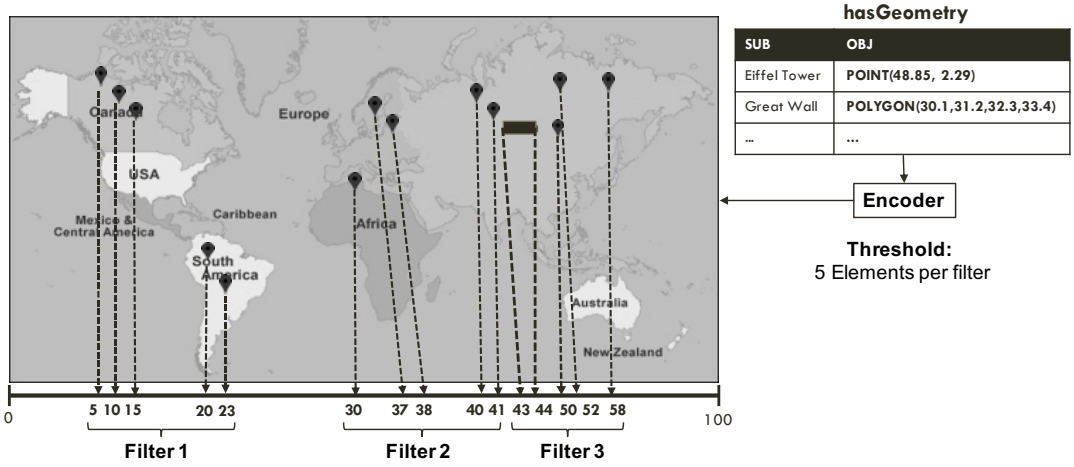

Fig. 5.2. Spatial encoding of RDF resources.

Figure 5.2 illustrates how RDF spatial data is encoded. The RDF triples objects of spatial predicates, *e.g.*`hasGeometry`, have literal objects containing spatial coordinates. The spatial coordinates represent *e.g.*, points or polygons. Each point is encoded using an encoder such as HSFC or Geohash. Finally, the encoded data is added to the appropriate filter using the triple subject (*i.e.*, resource name).

5.4.2 Temporal Encoding

Similar to spatial properties, temporal encoding allows describing time-related information for temporal predicates (*e.g.*, `hDate`, `bornIn`, and `diedIn`). Temporal encoding is used to represent the *temporal dimension* of RDF resources. The temporal encoding consists of 6 parts representing the year, month, day, hour, minute, and

seconds. The temporal data of any RDF object is cast to this encoding format to guarantee consistency of the encoded data.

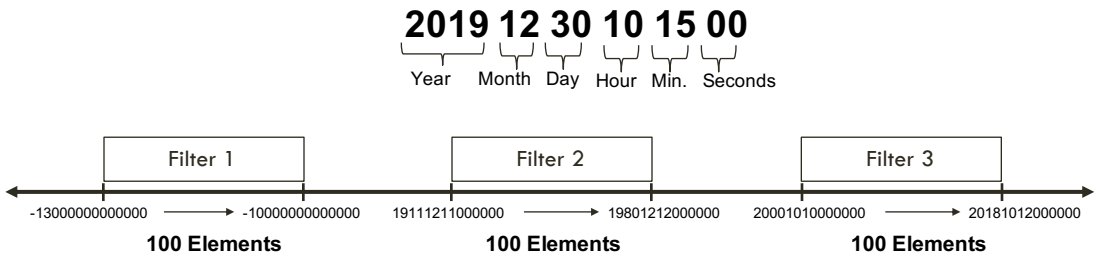

Fig. 5.3. Temporal encoding of RDF resources.

Figure 5.3 illustrates 300 encoded resource. The encoded data can be negative, *e.g.*prehistoric age, or positive, *e.g.*Gregorian calendar. The encoded data is then sorted based on its value. This generates a continuous timeline representing all the temporal resources. Finally, the range is split based on the count of resources, *e.g.*, every 100 elements. Every encoded element in a split is added to a single temporal filter by the resource name (*i.e.*, the subject of the triple).

5.4.3 Ontological Encoding

Similar to the spatial and temporal encoding, the objective of developing an ontological encoding scheme is to allow semantic filtering to be done on the ontological aspect of RDF resources. Unlike spatial and temporal encoding, there exists no direct method of reducing multiple ontological aspects of an RDF resource to a single, sortable dimension.

The first step in creating an ontological encoding is to build a hierarchical tree based on the `subClassOf` triples. Figure 5.4 illustrates a hierarchical tree that is built based on `subClassOf` triples that exist in a dataset. For example, the triples describing the `subClassOf` relations of a resource `Thing` can be seen as `Person`, `Animal`, and `Vehicle`.

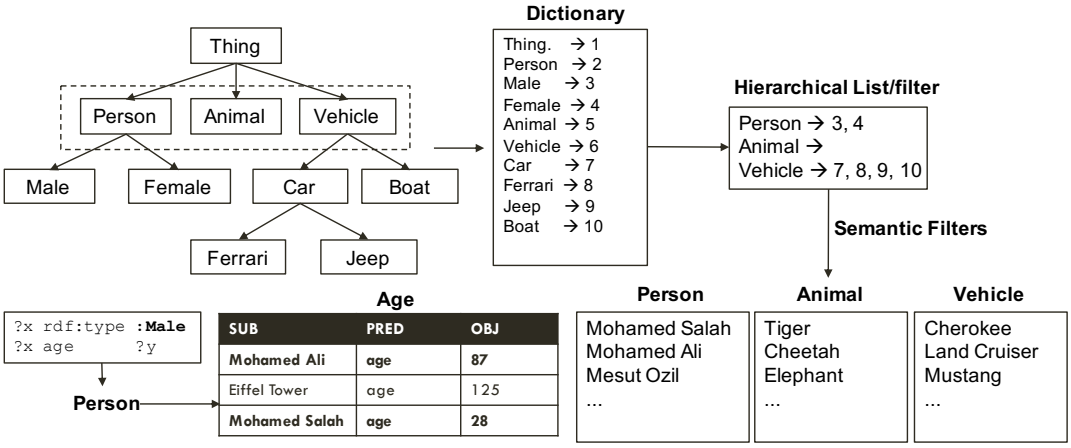

Fig. 5.4. Ontological encoding of RDF resources.

The second step is to encode every element in the hierarchy into a numeric representation. For example, the resource `Thing` is assigned the value 1. The numeric encoding allows adding resources to set membership structures, *e.g.*Roaring bitmaps, efficiently.

The third step is to represent the hierarchical structure at a specific level as a set of list structures. For example, if level 2 is chosen as the encoding level, then `Person`, `Animal`, and `Vehicle` will be defined as lists. Every list includes all the children of a resource. For example, `Vehicle` includes the encoding of every subclass element such as `Car`, `Boat`, `Ferrari`, and `Jeep` corresponding to numeric values 7, 8, 9 and 10 respectively.

The fourth step is to create a filter corresponding to every list created. The filters include all resources that have a triple pattern belonging to a specific ontological class. For example, all resources that have a `subClassOf` triple with an object being `Person`, `Male` or `Female` will be added to a filter corresponding to `Person`. The number of filters created corresponds to the number of ontological elements at the selected level in the hierarchy.

5.5 Indexing Semantic Filters

Once the semantic aspects of a query are determined, an encoding phase of every semantic aspect is performed to identify what filter to use. The filter is the indicator of whether specific resources exist given the query semantic aspects. However, this requires scanning all filters to determine which filter to probe. Having an index that represents the ranges covered by each filter can break the linear scan of filters, especially if the number and sizes of filters are large.

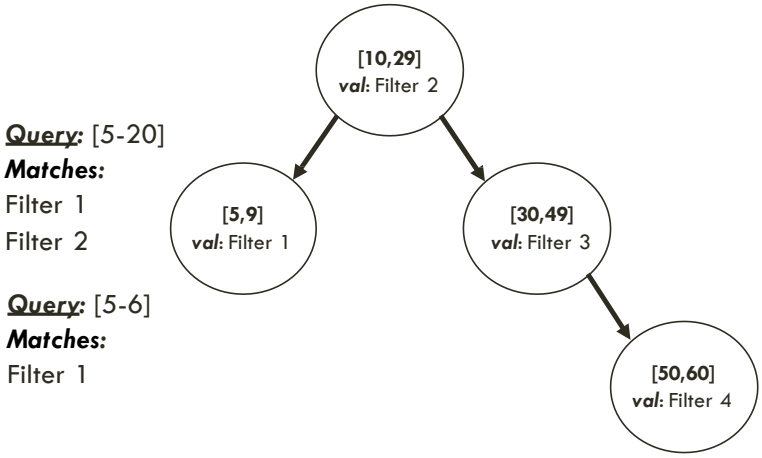

Fig. 5.5. An interval tree is used to index semantic filters.

Given an encoding of a query semantic aspect, the index is required to identify which semantic filters need to be used. An interval tree is one structure that satisfies this objective. Figure 5.5 illustrates an interval tree built for a set of filters where each node in the tree represents the range that the node covers and the filter that covers that range. For example, given a query where the encoded value ranges from 5 to 20, the interval tree presents Filter 1 and two as the ones that match that query range. Similarly, if the query range was 5 to 6 then Filter 1 will be the only match.

5.6 Query Processing using Semantic Filters

The objective of creating semantic filters is to allow reducing irrelevant resources from query processing as early as possible. This includes reducing the size of intermediate results in a distributed setting. Reducing irrelevant results in a distributed setting involves pushing down the filtering process to the node level. This requires that each node be aware of what the query semantic aspects are in order to determine which filters to use.

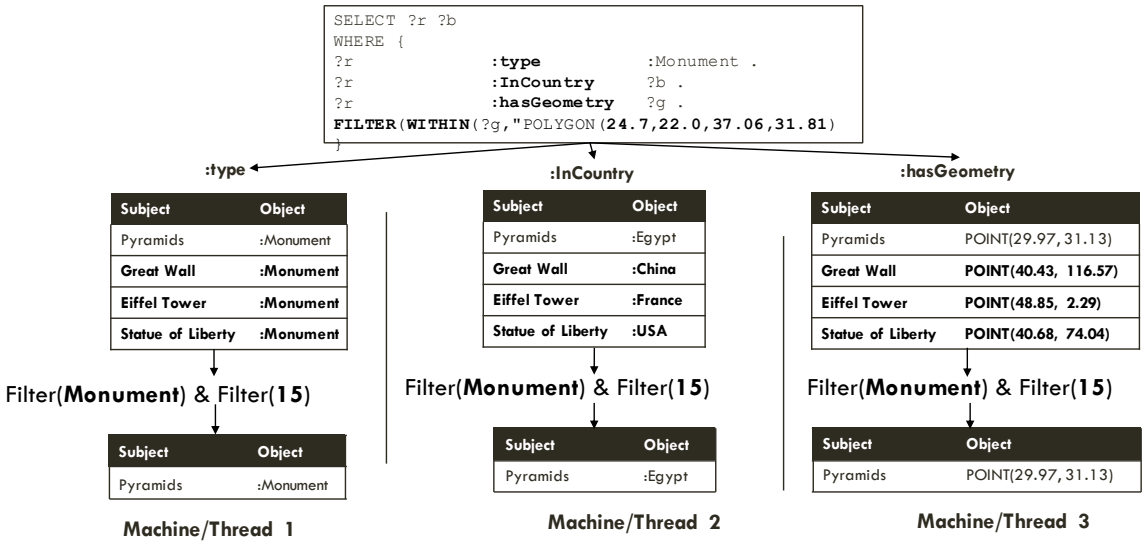

Fig. 5.6. Semantic filters applied in a distributed setting.

Figure 5.6 illustrates a query that has an ontological and spatial aspect. In specific, the query requires finding resources of type `Monument` within a specific spatial range. The first step required to determine which semantic filters to use is to encode all the semantic aspects of the query. For example, assuming that the ontological aspect had filters created at the same level as `Monument` then a filter with the same name will be used. For the spatial aspect, encoding the range query values specified in the query using the Geohash or HSFC techniques, *e.g.*, `Filter(15)` representing Egypt.

For both the spatial and ontological aspects, the interval tree can be used to find the matching filters to use, *e.g.*, `Filter(Monument)` and `Filter(15)`.

5.7 Experimental Evaluation

We implemented the semantic filtering over Knowledge Cubes or KC for short [3]. KC is an RDF query processing system that runs over Spark [69]. Realizing semantic filters on top of a Hadoop-based system demonstrates the applicability of the proposed techniques over widely adopted components, *e.g.*, HDFS, and Spark. Semantic filters can also be implemented in other specialized RDF query processing systems. Semantic filtering is compared against the baseline implementation of KC where no filtering is performed.

5.7.1 Experimental Setup

Our experiments are conducted using a real dataset, namely YAGO2s 2.5.3 [20, 21]. YAGO2s contains 245 million triples. YAGO2s benchmark queries are used to compare the query execution time [77]. The queries are defined by a semantic aspect or an RDF aspect. The RDF aspect refers to a SPARQL triple pattern while the semantic aspect refers to a query having a spatial, temporal, or ontological aspect. For each benchmark query, the first letter of the query name refers to the semantic aspect while the second letter refers to the RDF aspect. The (S) symbol refers to the aspect being selective (*i.e.*, Small) while the (L) refers to the aspect as being non-selective (*i.e.*Large).

All experiments are performed using a cluster consisting of 12 nodes. The cluster uses Cloudera 5.9. Spark 2 is used as a computational framework and Hadoop HDFS as a distributed file-system. Each node consists of 168 GB of RAM and eight cores. The total HDFS size is 10 Terabyte.

[3] `http://github.com/purduedb/knowledgecubes`

5.7.2 Experimental Results

The experiments measure various aspects of semantic filters including (1) number of generated filter per aspect, (2) storage overhead, (3) query execution performance, (4) reduction rate. KC is used as a baseline.

Spatial Aspect

The first set of experiments measure the performance of the spatial aspect. The spatial filters are created from the `hasLatitiude` and `hasLongitude` properties. The objects for triples having these two properties describe the spatial aspects of the corresponding subjects. For example, a resource such as `Egypt` will have spatial aspect triples, one describing the latitude and longitude.

Table 5.1.
Number of filters generated for 10K, 100K, 1M element per spatial aspect filter.

Count	Number of filters
10K	673
100K	68
1M	7

Table 5.1 illustrates the number of filters created given 10K, 100K, and 1M elements per filter. The number of triples representing the `hasLatitiude` and `hasLongitude` properties is 7 million triples. Using a filter size of 10K, the number of generated filters reaches 673. However, if the filter size is 1 million, then the total number of filters is 7.

Table 5.2 illustrates the number of sizes of filters generated given different sized filters. The smallest sizes are generated when the largest filters are used. The reason is that each generated filter can use hashing, *e.g.*Bloom filter, or bits, *e.g.*Roaring bitmaps, to represent the elements. This implies that adding elements to an existing

Table 5.2.
Sizes of filters generated for 10K, 100K, 1M element per spatial aspect filter.

Count	Size (MB)
10K	18
100K	15
1M	11

structure is less expensive than creating a new structure with a smaller size. However, this is not always true if the used structure is probabilistic in which the false positive rate will increase.

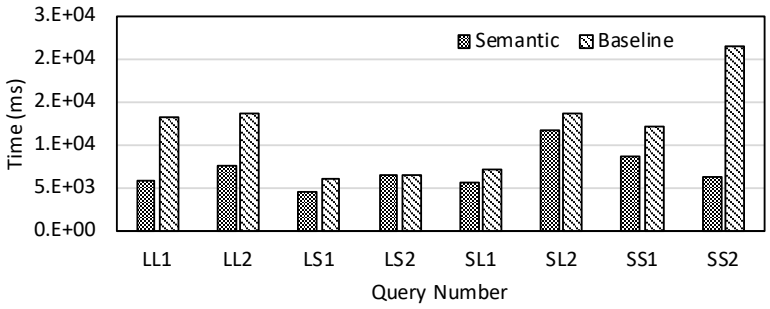

Fig. 5.7. Query processing performance using the spatial aspect.

Figure 5.7 illustrates the query processing performance when using filters of size 1 million to process the benchmark queries. The results for queries LL1, LL2, and SS2 achieve less than half the query processing overhead compared to the baseline.

Figure 5.8 illustrates the reduction rate achieved when considering the spatial aspect of the query. The spatial aspect provides an opportunity to remove almost an order of magnitude of irrelevant RDF resources for non-selective queries. Also, the reduction rate is also significant when the spatial aspect is non-selective while the RDF aspect is selective.

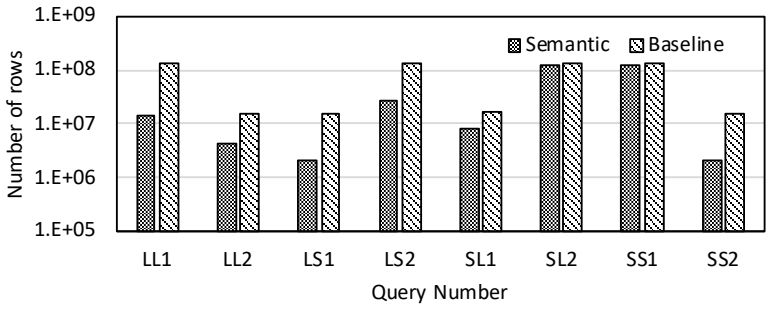

Fig. 5.8. Reduction rate using the spatial aspect.

Table 5.3.
Number of filters generated per query for 10K, 100K, 1M element per spatial aspect filter.

Query	**10K**	**100K**	**1M**
LL1	59	7	1
LL2	156	17	2
LS1	1	1	1
LS2	93	10	2
SL1	203	21	3
SL2	184	19	3
SS1	6	2	1
SS2	12	2	1

Table 5.3 illustrates the number of filters used when processing every benchmark query based on different filter sizes. The results illustrate how using smaller filters forces the query processor to use more filters during the filtering process. This also translates to lower efficiency in terms of query processing performance.

Figure 5.9 illustrates the query processing performance given different-sized spatial aspect filters. The results illustrate the degradation of query processing performance when the filters are small. This leads to a larger number of filters to be used as shown in Table 5.3.

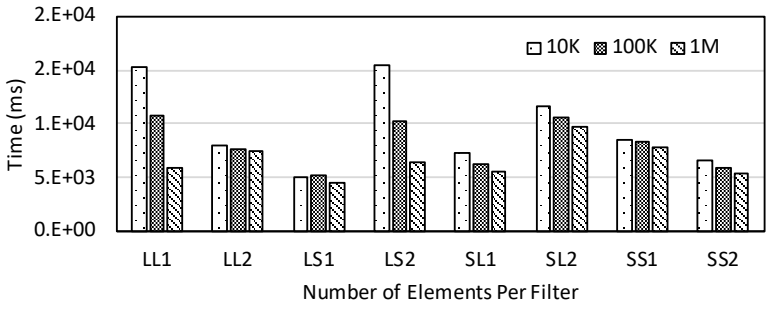

Fig. 5.9. Query processing performance using 10K, 100K, and 1M elements per spatial aspect filter.

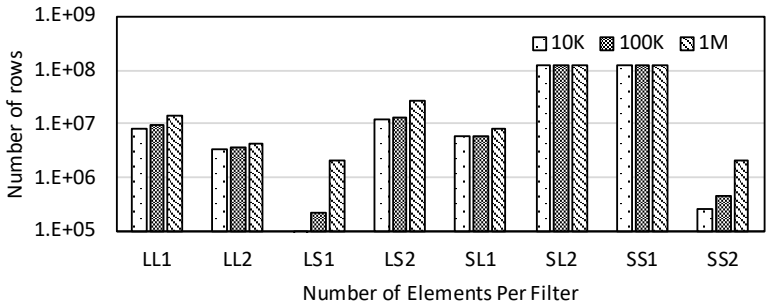

Fig. 5.10. Reduction rate using using 10K, 100K, and 1M elements per spatial aspect filter.

Figure 5.10 illustrates the advantage of having smaller filters with respect to the reduction rate. In contrast to the advantage of having fewer filters used during query processing, having a more comprehensive coverage of resources when using smaller filters has a small advantage with respect to the reduction rate.

Temporal Aspect

The second set of experiments measure the performance of the temporal aspect. The temporal filters are created for `wasBornOnDate` and `wasCreatedOnDate` proper-

ties. The objects for triples having these two properties describe the temporal aspects of the corresponding subjects. For example, a resource such as `Mohamed Ali` will have temporal aspect triples using the property `wasBornOnDate` while a resource such as `Pyramids` will have temporal aspect triples using the property `wasCreatedOnDate`.

Table 5.4.
Number of filters generated for 1K, 5K, 50K element per temporal aspect filter.

Count	**Number of filters**
1K	31
5K	7
50K	3

Table 5.4 illustrates the number of temporal filters created when creating filters of sizes 1K, 5K, and 50K. The temporal coverage for the selected predicates (*i.e.*, `wasBornOnDate` and `wasCreatedOnDate`) is smaller than the spatial coverage in the YAGO dataset where `wasBornOnDate` contains 686,072 triples while `wasCreatedOnDate` contains 507,733 triples. Similar to the spatial aspect filters, the smaller the size of the filter, the larger number of filters are created.

Table 5.5.
Sizes of filters generated for 1K, 5K, 50K element per temporal aspect filter.

Count	**Size (MB)**
1K	1.2
5K	1.1
50K	1

Table 5.5 illustrates the sizes of generated filters. Given the smaller number of resources contained within each filter, the overall sizes for the generated filters corresponding to `wasBornOnDate` and `wasCreatedOnDate` does not surpass 2MB in size.

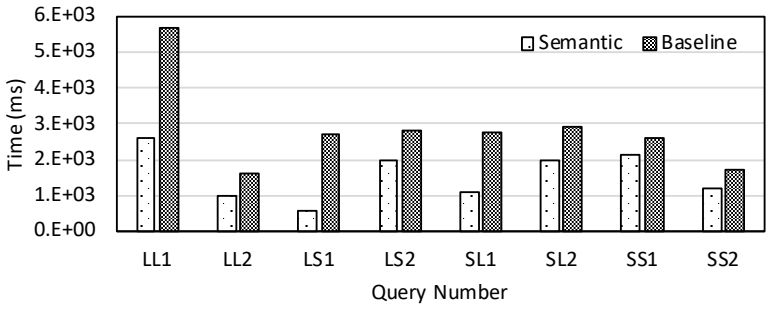

Fig. 5.11. Query processing performance using the temporal aspect.

Figure 5.11 illustrates the query processing performance when using the temporal aspect filters. The results illustrate how the benchmarks queries having a dominant RDF result set achieve almost an order of magnitude better results than the baseline. This demonstrates how selective the temporal dimension is and how effective it is as a criteria for filtering irrelevant resources from query processing.

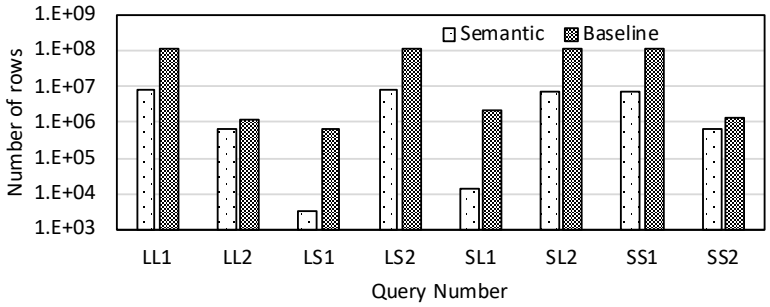

Fig. 5.12. Reduction rate using the temporal aspect.

Figure 5.12 illustrates the reduction rate achieved when using the temporal aspect filters. The results illustrate how the temporal dimension is effective in reducing irrelevant RDF resources. The reduction rate appears higher than the spatial aspect as less number of resources possess a temporal dimension, and thus the filtering process is more aggressive.

Table 5.6.
Number of filters generated per query for 1K, 5K, 50K element per temporal aspect filter.

Query	1K	5K	50K
LL1	25	6	1
LL2	25	6	1
LS1	15	4	2
LS2	25	6	1
SL1	24	6	2
SL2	25	6	2
SS1	8	3	1
SS2	12	3	2

Table 5.6 illustrates the number of filters used when answering every benchmark query. The number of filters used is less as the number of resources available per filter increases. In the worst case the total number of filters used if the filter size is 1K can reach 25 filters. This affects the overall query processing performance as shown in Figure 5.13 compared to using larger-sized filters.

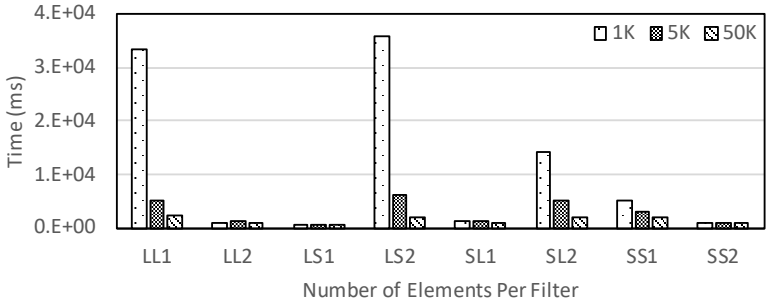

Fig. 5.13. Query processing performance using 1K, 5K, and 50K elements per temporal aspect filter.

Figure 5.13 illustrates the query processing performance across multiple filter sizes. The results illustrate how filters of size 1K significantly affect query processing performance when compared with the 50K sized filters. On the other hand, query `LL2`

exhibits better performance, unlike the other benchmark queries. The reason can be attributed to Spark as it handles smaller sized data efficiently. This is also evident in Figure 5.14 where the smallest sized relations among all benchmark queries is also LL2.

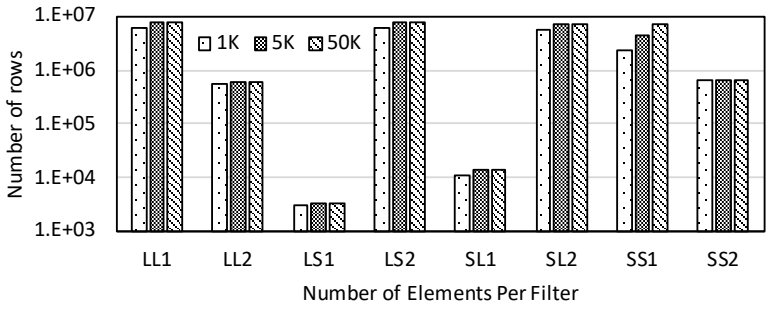

Fig. 5.14. Reduction rate using using 1K, 5K, and 50K elements per temporal aspect filter.

Figure 5.14 illustrates the reduction rate achieved for the temporal aspect using filters of 1K, 5K, 50K in size. The reduction rate is comparable across the three sizes.

Ontological Aspect

The third set of experiments measure the performance of the ontological aspect. The ontological filters are created at the first subclass level after `Thing` resource. The number of subclasses of `Thing` for the YAGO dataset is seven. Each resource belonging to a subclass of the seven top-level resources is added to the corresponding filter of the top-level resources. For this set of experiments, we used the benchmark queries proposed by [9] and added an additional triple that corresponds to an ontological aspect, namely `rdf:type` where we filter based on resources belonging to the `wordnet_person_100007846` class.

Table 5.7.
Number of filters generated per query for 1K, 5K, 50K element per temporal aspect filter.

Query	1K
wordnet_person_100007846	8997551
wordnet_organization_108008335	2137041
wordnet_building_102913152	2394753
yagoGeoEntity	38096604
wordnet_artifact_100021939	6836860
wordnet_abstraction_100002137	6575470
wordnet_physical_entity_100001930	27944652

Table 5.7 summarizes the number of resources that maps to each direct subclass of `Thing`. Each resource is recursively added to the appropriate filter based on the subclass relation with the top-level seven resources.

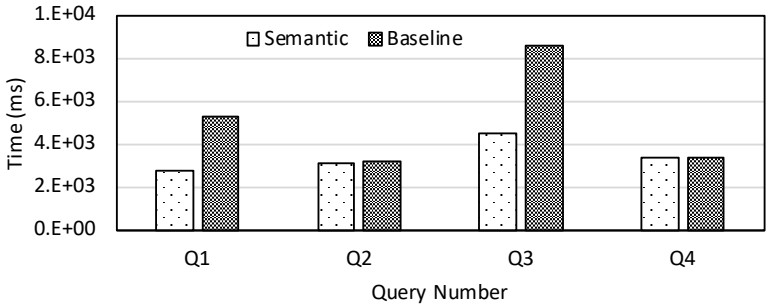

Fig. 5.15. Query processing performance using the ontological aspect.

Figure 5.15 illustrates the query processing performance when using the ontological aspect filters. The results illustrate how `Q1` and `Q3` achieve more than an order of magnitude better performance when utilizing the ontological aspect for filtering the base tables. On the other hand, `Q2` and `Q3` achieve comparable performance. This is attributed to Spark where smaller-sized relations are executed more efficiently.

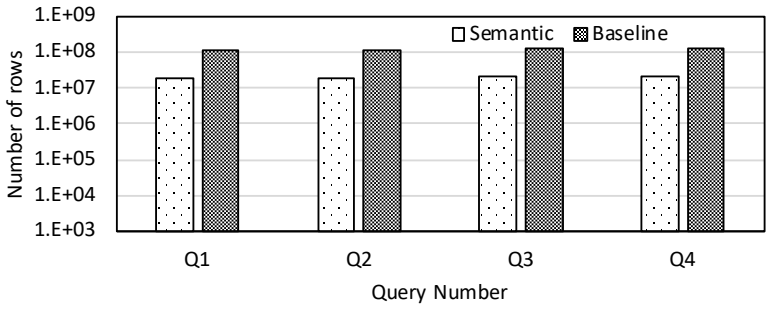

Fig. 5.16. Reduction rate using the ontological aspect.

Figure 5.16 illustrates the reduction rate achieved when using the ontological aspect filters. The results illustrate how the ontological aspect achieves nearly an order-of-magnitude reduction rate when compared to the base relations.

5.8 Concluding Remarks

In this chapter, we introduce semantic filtering. First, we introduce how semantic filtering can be used to efficiently filter data based on the semantic aspects of the query. Second, we introduce a set of encoding schemes for queries possessing a spatial, temporal, and ontological aspect. Finally, we introduce a method for using semantic filters to perform efficient distributed query processing of queries by filtering irrelevant RDF resources. The results demonstrate how semantic filters on top of Spark achieves an order of magnitude improvement in terms of query performance compared to a baseline implementation.

6. CONCLUSIONS

In this dissertation, we study efficient query processing over web-scale RDF data. In Chapter 1, we show how semantic data is becoming an integral component for nowadays demanding and sophisticated user applications and search engines that provide answers beyond keyword-based matches in support of the Semantic Web. We showcase how RDF is becoming the defacto standard for semantic data representation in the Semantic Web. We discuss how the astronomical growth of RDF data calls for scalable RDF management and query processing strategies.

Chapter 2 presents RDF query optimization techniques over vertically partitioned (VP) data. Bloom join is presented as a technique or computing a reduced sets of RDF triples that satisfy specific join pattern(s). We study the effect of caching reductions rather than caching the final results. Also, we study partitioning of RDF data triples using the join attributes of the query instead of using a static criteria. Also, we present how to efficiently answer unbound property queries using Bloom filters. Extensive experimentation using synthetic and real datasets demonstrate an order of magnitude enhancement in terms of preprocessing time, storage, and query performance.

Chapter 3 presents SPARTI, a semantic partitioning technique for RDF data. We introduce SemVP as a relational partitioning scheme that provides row-level semantics for RDF data. The row-level semantics allow reading a reduced set of rows when answering specific query join-patterns. We introduce an algorithm for discovering co-occurring properties in the query-workload and use the co-occurring properties to compute join filters. A cost-model for managing join filters is proposed and is used to prioritizes the creation of the important join filters. The experimental study shows how SPARTI achieves robust performance over synthetic and real datasets compared to state-of-the-art cloud-based solutions.

Chapter 4 presents KC, an RDF query processing system for cloud-based platforms. KC adopts a generalized filtering approach where exact or approximate set membership structures can be used by defining a set of common operations that both support. Generalized filtering is used for reducing non-matching join entries in an online fashion. Extensive experimentation using synthetic and real datasets demonstrate how KC achieves an order of magnitude improvement in terms of preprocessing time, storage, and query performance compared to a cloud-based state-of-the-art system.

Finally, Chapter 5 presents semantic filtering, a technique for efficiently filtering data based on the semantic aspects of the query. The semantic aspects can be spatial, temporal, or ontological. We introduce a set of encoding schemes that targets each semantic aspect. Also, we introduce a method for utilizing semantic filters in a distributed fashion. The results demonstrate how semantic filters allow query processing systems to achieve an order of magnitude improvement compared to a baseline implementation.

REFERENCES

REFERENCES

[1] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang, "Knowledge vault: a web-scale approach to probabilistic knowledge fusion," in *KDD*, M. Kaeberlein, Ed., vol. 7, no. 5. New York, New York, USA: ACM Press, 5 2014, pp. 601–610.

[2] E. K. Neumann and D. Quan, "BioDash: a Semantic Web dashboard for drug development." *Pacific Symposium on Biocomputing*, pp. 176–87, 2006.

[3] 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, 2 2010.

[4] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "TripleBit: a fast and compact system for large scale RDF data," *Proceedings of the VLDB Endowment*, vol. 6, no. 7, pp. 517–528, 5 2013.

[5] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao, "gStore: a graph-based SPARQL query engine," *The VLDB Journal*, vol. 23, no. 4, pp. 565–590, 8 2014.

[6] J. H. And, D. J. A. And, and K. Ren, "Scalable SPARQL Querying of Large RDF Graphs," *PVLDB*, vol. 4, pp. 1123–1134, 8 2011.

[7] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale RDF data," *VLDB*, pp. 265–276, 2 2013.

[8] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, "S2RDF," *Proceedings of the VLDB Endowment*, vol. 9, no. 10, pp. 804–815, 6 2016.

[9] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis, "A survey and experimental comparison of distributed SPARQL engines for very large RDF data," *Proceedings of the VLDB Endowment*, vol. 10, no. 13, pp. 2049–2060, 9 2017.

[10] Z. Kaoudi and I. Manolescu, "RDF in the clouds: a survey," *The VLDB Journal*, vol. 24, no. 1, pp. 67–91, 2 2015.

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

[12] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds, "Efficient RDF storage and retrieval in Jena2," *International Workshop on Semantic Web and Databases*, pp. 35–43, 2003.

[13] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan, "An Efficient SQL-based RDF Querying Scheme," *VLDB 2005, Trondheim, Norway, August 30 - September 2*, pp. 1216–1227, 2005.

[14] L. Rietveld, R. Hoekstra, S. Schlobach, and C. Guéret, "Structural Properties as Proxy for Semantic Relevance in RDF Graph Sampling," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 81–96, 10 2014.

[15] L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for local queries," *ACM SIGMOD Record*, vol. 15, no. 2, pp. 84–95, 1986.

[16] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[17] A. Madkour, W. G. Aref, and S. Basalamah, "Knowledge cubes : A proposal for scalable and semantically-guided management of Big Data," in *2013 IEEE International Conference on Big Data*. IEEE, 10 2013, pp. 1–7.

[18] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified Stress Testing of RDF Data Management Systems," *ISWC*, pp. 197–212, 2014.

[19] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Web Semantics*, pp. 158–182, 2005.

[20] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum, "YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia," *Artificial Intelligence*, vol. 194, pp. 28–61, 1 2013.

[21] J. Biega, E. Kuzey, and F. M. Suchanek, "Inside YAGO2s: a transparent information extraction architecture," in *Proceedings of the 22nd International Conference on World Wide Web - WWW '13 Companion*. New York, New York, USA: ACM Press, 2013, pp. 325–328.

[22] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM*, pp. 359–392, 1998.

[23] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris, "H2RDF+: High-performance distributed joins over large-scale RDF graphs," in *2013 IEEE International Conference on Big Data*. IEEE, 10 2013, pp. 255–263.

[24] K. Lee and L. Liu, "Scaling queries over big RDF graphs with semantic hash partitioning," *VLDB*, pp. 1894–1905, 9 2013.

[25] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing," in *SIGMOD*. New York, New York, USA: ACM Press, 2014, pp. 289–300.

[26] B. Wu, Y. Zhou, P. Yuan, L. Liu, and H. Jin, "Scalable SPARQL querying using path partitioning," in *ICDE*, 2015, pp. 795–806.

[27] P. Peng, L. Zou, L. Chen, and D. Zhao, "Query Workload-based RDF Graph Fragmentation and Allocation," *EDBT*, pp. 377–388, 2016.

[28] T. Rabl and H.-A. Jacobsen, "Query Centric Partitioning and Allocation for Partially Replicated Database Systems," in *SIGMOD*. New York, New York, USA: ACM Press, 2017, pp. 315–330.

[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*, vol. 9, no. 7, pp. 564–575, 3 2016.

[30] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins," in *ICDE*. IEEE, 4 2011, pp. 984–994.

[31] M. Meimaris, G. Papastefanatos, N. Mamoulis, and I. Anagnostopoulos, "Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization," in *ICDE*. IEEE, 4 2017, pp. 497–508.

[32] R. Castillo and U. Leser, "Selecting Materialized Views for RDF Data," in *ICWE*, 2010, pp. 126–137.

[33] N. Papailiou, Dimitrios Tsoumakos, P. Karras, and N. Koziris, "Graph-Aware , Workload-Adaptive SPARQL Query Caching," *SIGMOD*, pp. 1777–1792, 2015.

[34] M. Yang and G. Wu, "Caching intermediate result of SPARQL queries," in *WWW*. New York, New York, USA: ACM Press, 2011, p. 159.

[35] 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*, vol. 44, no. 2, pp. 439–474, 8 2015.

[36] P. Ravindra and K. Anyanwu, "Scaling Unbound-Property Queries on Big RDF Data Warehouses using MapReduce," in *EDBT*, 2015, pp. 169–180.

[37] G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis, and D. Plexousakis, "RDF query answering using apache spark: Review and assessment," in *ICDEW*, 2018, pp. 54–59.

[38] K. Rohloff and R. E. Schantz, "High-performance, massively scalable distributed systems using the MapReduce software framework," *PSIEtA*, pp. 1–5, 2010.

[39] P. a. Bernstein and D.-M. W. Chiu, "Using Semi-Joins to Solve Relational Queries," *Journal of the ACM*, pp. 25–40, 1981.

[40] D. C. Faye, O. Curé, and G. Blin, "A survey of RDF storage approaches," *HAL Archives*, vol. 15, pp. 11–35, 2012.

[41] M. T. Özsu, "A survey of RDF data management systems," *Frontiers of Computer Science*, vol. 10, no. 3, pp. 418–432, 6 2016.

[42] S. Sakr and G. Al-Naymat, "Relational processing of RDF queries," *ACM SIGMOD Record*, vol. 38, no. 4, p. 23, 6 2010.

[43] P. Cudre-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. L. Keppmann, D. Miranker, J. F. Sequeda, and M. Wylot, "NoSQL databases for RDF: An empirical evaluation," *ISWC*, pp. 310–325, 2013.

[44] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)," *VLDB*, vol. 3, no. 1-2, pp. 515–529, 2010.

[45] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad, "Only aggressive elephants are fast elephants," *PVLDB*, vol. 5, no. 11, pp. 1591–1602, 2012.

[46] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *OSDI*, pp. 137–149, 2004.

[47] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen, "PigSPARQL: Mapping SPARQL to Pig Latin," *SWIM*, pp. 1–8, 2011.

[48] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A Not-So-Foreign Language for Data Processing," *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, p. 1099, 2008.

[49] H. Kim, P. Ravindra, and K. Anyanwu, "From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra," *VLDB*, vol. 4, no. 12, pp. 1426–1429, 2011.

[50] M. F. Husain, J. McGlothlin, M. M. Masud, L. R. Khan, and B. Thuraisingham, "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing," *TKDE*, pp. 1312–1327, 2011.

[51] J. H. Du, H. F. Wang, Y. Ni, and Y. Yu, "HadoopRDF: A scalable semantic data analytical engine," in *ICIC*. Springer, Berlin, Heidelberg, 2012, pp. 633–641.

[52] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder, "Impala: A Modern, Open-Source SQL Engine for Hadoop," *CIDR*, 3 2015.

[53] M. Armbrust, A. Ghodsi, M. Zaharia, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, and M. J. Franklin, "Spark SQL: Relational Data Processing in Spark," in *SIGMOD*. New York, New York, USA: ACM Press, 2015, pp. 1383–1394.

[54] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient RDF store over a relational database," in *SIGMOD*. New York, New York, USA: ACM Press, 6 2013, p. 121.

[55] A. Schätzle, M. Przyjaciel-zablocki, A. Neu, and G. Lausen, "Sempala: Interactive SPARQL Query Processing on Hadoop," *ISWC*, pp. 164–179, 2014.

[56] C. Weiss, P. Karras, and A. Bernstein, "Hexastore : Sextuple Indexing for SemanticWeb Data Management," *VLDB*, vol. 1, no. 1, pp. 1008–1019, 8 2008.

[57] R. Punnoose, A. Crainiceanu, and D. Rapp, "Rya," in *Proceedings of the 1st International Workshop on Cloud Intelligence - Cloud-I '12*. New York, New York, USA: ACM Press, 2012, pp. 1–8.

[58] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris, "H2RDF+: an efficient data management system for big RDF graphs," in *SIGMOD*. New York, New York, USA: ACM Press, 2014, pp. 909–912.

[59] T. Neumann and G. Weikum, "Scalable join processing on very large RDF graphs," *SIGMOD*, pp. 627–639, 2009.

[60] K. Hose and R. Schenkel, "WARP: Workload-aware replication and partitioning for RDF," *ICDE*, pp. 1–6, 2013.

[61] L. Galárraga and R. Schenkel, "Partout : A Distributed Engine for Efficient RDF Processing," *WWW*, pp. 267–268, 2014.

[62] M. Hammoud, D. A. Rabbou, R. Nouri, SeyedMehdiReza, Beheshti, and Sherif Sakr, "DREAM : Distributed RDF Engine with Adaptive Query Planner and Minimal Communication," *VLDB*, pp. 654–665, 2015.

[63] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: a column-oriented DBMS," *VLDB*, pp. 553–564, 2005.

[64] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach, "SW-Store: A vertically partitioned DBMS for semantic web data management," *VLDB*, vol. 18, no. 2, pp. 385–406, 2009.

[65] M. T. Özsu and P. Valduriez, *Principles of distributed database systems, third edition*. Springer-Verlag New York, 2011.

[66] V. Raman, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, L. Zhang, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, and S. Liu, "DB2 with BLU acceleration," *VLDB*, vol. 6, no. 11, pp. 1080–1091, 2013.

[67] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin, "Fine-grained partitioning for aggressive data skipping," *SIGMOD*, pp. 1115–1126, 2014.

[68] A. W. Bowman and A. Azzalini, *Applied Smoothing Techniques for Data Analysis*. OUP Oxford, 1 1997.

[69] M. Zaharia, M. Chowdhury, T. Das, and A. Dave, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *Nsdi '12*, pp. 2–2, 2012.

[70] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer, and C. Bizer, "DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia," *Semantic Web*, pp. 167–195, 2015.

[71] M. Przyjaciel-Zablocki, A. Schaetzle, E. Skaley, T. Hornung, and G. Lausen, "Map-Side Merge Joins for Scalable SPARQL BGP Processing," in *CloudCom*. IEEE, 12 2013, pp. 631–638.

[72] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with Roaring bitmaps," *Software - Practice and Experience*, vol. 46, no. 5, pp. 709–719, 2016.

[73] F. Belleau, M. A. Nolin, N. Tourigny, P. Rigault, and J. Morissette, "Bio2RDF: Towards a mashup to build bioinformatics knowledge systems," *Journal of Biomedical Informatics*, vol. 41, no. 5, pp. 706–716, 10 2008.

[74] A. Madkour, A. M. Aly, and W. G. Aref, "WORQ: Workload-Driven RDF Query Processing," in *ISWC*, 2018, pp. 583–599.

[75] T. White, *Hadoop: The definitive guide*. Yahoo Press, 2015, vol. 54.

[76] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, "Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning," *VLDB*, pp. 1–26, 2016.

[77] J. Liagouris, N. Mamoulis, P. Bouros, and M. Terrovitis, "An Effective Encoding Scheme for Spatial RDF Data," *VLDB*, pp. 1271–1282, 2014.

[78] K. Kyzirakos, M. Karpathiotakis, and M. Koubarakis, "Strabon: A semantic geospatial DBMS," *Lecture Notes in Computer Science*, vol. 7649 LNCS, no. PART 1, pp. 295–311, 2012.

[79] D. Wang, L. Zou, Y. Feng, X. Shen, J. Tian, and D. Zhao, "S-store: An engine for large RDF graph integrating spatial information," *DASFAA*, vol. 7826 LNCS, no. PART 2, pp. 31–47, 2013.

[80] R. Battle and D. Kolas, "Enabling the Geospatial SemanticWeb with Parliament and GeoSPARQL," *Semantic Web Journal*, vol. 0, no. 0, pp. 1–17, 2012.

[81] M. Koubarakis and K. Kyzirakos, "Modeling and querying metadata in the semantic sensor web: The model stRDF and the query language stSPARQL," in *LNCS*, vol. 6088 LNCS, no. PART 1, 2010, pp. 425–439.

[82] J. Broekstra, A. Kampman, and Frank van Harmelen, "Sesame: An architecture for storing and querying RDF data and schema information," in *The First International Semantic Web Conference on The Semantic Web*, 2002, pp. 54–68.

[83] A. Brodt, D. Nicklas, and B. Mitschang, "Deep integration of spatial query processing into native RDF triple stores," in *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS '10*. New York, New York, USA: ACM Press, 2010, p. 33.

[84] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gStore: : Answering SPARQL Queries via Subgraph Matching," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 482–493, 5 2011.

[85] A. Eldawy and M. F. Mokbel, "SpatialHadoop: A MapReduce framework for spatial data," in *Proceedings - International Conference on Data Engineering*, vol. 2015-May, 2015, pp. 1352–1363.

[86] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "LocationSpark," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1565–1568, 9 2016.

[87] D. Xie, F. Li, B. Yao, G. Li, Z. Chen, L. Zhou, and M. Guo, "Simba," in *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS '16*, 2016, pp. 1–4.

[88] J. Yu, J. Wu, and M. Sarwat, "A demonstration of GeoSpark: A cluster computing framework for processing big spatial data," in *2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016*, 2016, pp. 1410–1413.

VITA

VITA

Amgad Madkour received his bachelor's degree in computer science from the Arab Academy for Science and Technology in 2003. He received his master's degree in computers science from the American University in Cairo in 2009. He joined the Ph.D. program at Purdue University in fall 2010. His research interests are in the area of data management where he focuses on efficient query processing over semantic data. Before joining Purdue, he was a Staff Research Engineer in the human language technologies group at IBM Egypt. As a graduate student, he interned at Microsoft Research, Microsoft AI and Research (AI+R), and Yahoo!. He received his Ph.D. in computer science from Purdue in fall 2018. He joined Microsoft AI and Research (AI+R) as a Data Scientist to work on the Microsoft Satori Knowledge graph.

PUBLICATIONS

- **Amgad Madkour**, Ahmed M. Aly, Walid G. Aref, "WORQ: Workload-Driven RDF Query Processing," International Semantic Web Conference (ISWC) 2018, pp. 583-599
- **Amgad Madkour**, Walid G. Aref, Ahmed M. Aly, "SPARTI: Scalable RDF Data Management Using Query-Centric Semantic Partitioning," SBD@SIGMOD, pp. 1:1-1:6, 2018.
- **Amgad Madkour**, Walid G. Aref, Sunil Prabhakar, Mohamed H. Ali, Siarhei Bykau, "TrueWeb: A Proposal for Scalable Semantically-Guided Data Management and Truth Finding in Heterogeneous Web Sources," SBD@SIGMOD, pp. 5:1-5:6, 2018.
- **Amgad Madkour**, Walid G. Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, Saleh M. Basalamah, "A Survey of Shortest-Path Algorithms," CoRR abs/1705.02044, 2017.
- Ahmed R. Mahmood, Ahmed M. Aly, Thamir Qadah, El Kindi Rezig, Anas Daghistani, **Amgad Madkour**, Ahmed S. Abdelhamid, Mohamed S. Hassan, Walid G. Aref, Saleh M. Basalamah, "Tornado: A Distributed Spatio-Textual Stream Processing System," PVLDB 8(12), pp. 2020-2023, 2015.

- **Amgad Madkour**, Walid G. Aref, Mohamed F. Mokbel, Saleh M. Basalamah, "Geo-tagging non-spatial concepts," MobiGIS@SIGSPATIAL, pp. 31-39, 2015.
- **Amgad Madkour**, Walid G. Aref, Saleh M. Basalamah, "Knowledge cubes - A proposal for scalable and semantically-guided management of Big Data," IEEE BigData, pp. 1-7, 2013.
- Eduard C. Dragut, Peter Baker, Jia Xu, Muhammad I. Sarfraz, Elisa Bertino, **Amgad Madkour**, Raghu Agarwal, Ahmed R. Mahmood, Sangchun Han, "CRIS - Computational research infrastructure for science," IRI, pp. 301-308, 2013.
- Eduard C. Dragut, Mourad Ouzzani, **Amgad Madkour**, Nabeel Mohamed, Peter Baker, David E. Salt, "Ionomics Atlas: a tool to explore interconnected ionomic, genomic and environmental data," CIKM, pp. 2680-2682, 2012.
- Sara Noeman, **Amgad Madkour**, "Language Independent Transliteration Mining System Using Finite State Automata Framework," NEWS@ACL, pp. 57-61, 2010.
- **Amgad Madkour**, Ahmed Sameh, "Intelligent Open Spaces: Using Neural Networks for Prediction of Requested Resources in Smart Spaces," CSE, pp. 132-138, 2008.
- **Amgad Madkour**, Sherif G. Aly, "Resource Sharing Systems: A Combinatorial Application to Pervasive Computing," AICCSA, pp. 153-157, 2007.
- **Amgad Madkour**, Kareem Darwish, Hany Hassan, Ahmed Hassan, Ossama Emam, "BioNoculars: Extracting Protein-Protein Interactions from Biomedical Text," BioNLP@ACL, pp. 89-96, 2007.
- **Amgad Madkour**, Sherif G. Aly, "Pervasive Open Spaces: A Transparent and Scalable Dome-Based Pervasive Resource Allocation System," ISPA Workshops, pp. 115-124, 2006.
- Kareem Darwish, **Amgad Madkour**, "The GUC Goes to TREC 2004: Using Whole or Partial Documents for Retrieval and Classification in the Genomics Track," TREC 2004.

PATENTS

- **Amgad Madkour**, Saleh Basalamah, SYSTEM, DEVICE, AND METHOD FOR TRACKING PRAYER, United States Patent, Filed 25 September 2014.
- **Amgad Madkour**, Hany Hassan, SELECTING A DATA ELEMENT IN A NETWORK, United States Patent, Filed 25 July 2011.