SPARTI: Scalable RDF Data Management Using Query-Centric Semantic Partitioning

Amgad Madkour
Purdue University
West Lafayatte, IN, USA
amgad@cs.purdue.edu

Walid G. Aref
Purdue University
West Lafayatte, IN, USA
aref@cs.purdue.edu

Ahmed M. Aly
Google Inc.
Mountain View, CA, USA
aaly@google.com

ABSTRACT

Semantic data is an integral component for search engines that provide answers beyond simple keyword-based matches. Resource Description Framework (RDF) provides a standardized and flexible graph model for representing semantic data. The astronomical growth of RDF data raises the need for scalable RDF management strategies. Although cloud-based systems provide a rich platform for managing large-scale RDF data, the shared storage provided by these systems introduces several performance challenges, e.g., disk I/O and network shuffling overhead. This paper investigates SPARTI, a scalable RDF data management system. In SPARTI, the partitioning of the data is 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 utilizes a partitioning schema, termed SemVP, that enables the system to read a reduced set of rows instead of entire partitions. SPARTI proposes a budgeting mechanism with a cost model to determine the worthiness of partitioning. Using real and synthetic datasets, SPARTI is compared against a Spark-based state-of-the-art system and is shown to execute queries around half the time over all query shapes while maintaining around an order of magnitude enhancement in storage requirements.

CCS CONCEPTS

• Information systems → Database query processing; **Resource Description Framework (RDF)**;

KEYWORDS

RDF Data Management, Query Processing, Bloom Join, Vertical Partitioning, Hadoop, Spark

ACM Reference Format:
Amgad Madkour, Walid G. Aref, and Ahmed M. Aly. 2018. SPARTI: Scalable RDF Data Management Using Query-Centric Semantic Partitioning. In *SBD'18: Semantic Big Data , June 10, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3208352.3208356

1 INTRODUCTION

RDF is an integral component in many areas, e.g., Semantic Search, (e.g., Google Knowledge Vault [10]) , Smart governments, and medical systems. Distributed and cloud platforms, e.g., Hadoop [1], provide shared memory, storage, 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 data partitioning (both on disk and in memory) can significantly improve the query performance over these platforms.

An important observation is that not all entries of a partition are part of the final result of a query. In addition, only a small fraction of partitions are actually accessed in a real query-workload setting [18].

This paper investigates Semantic Partitioning (SPARTI, for short). SPARTI includes three phases for processing RDF queries (1) *property-based partitioning*, (2) *partition reduction*, and (3) *query processing*. In the *property-based partitioning* phase, SPARTI employs a relational partitioning schema, namely SemVP (Short for Semantic Vertical Partitioning). Similar to *Vertical Partitioning (VP)* [1], 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 *semantic filters*. The semantic filters indicate whether an entry in a partition is part of a query join result or not.

In the *partition reduction* phase, the semantic filters are continuously computed based on the query-workload. First, SPARTI uses the query-workload to identify the related (i.e. co-occurring) and frequently mentioned properties. Then, a set of semantic filters are created between every property and all its related properties. The semantic filters are computed using Bloom-join [16]. The join result represents the reduced set of rows that qualify a join pattern and is materialized as a semantic filter. In the *query processing* phase, SPARTI identifies the properties and join patterns in queries, and matches every property and query join pattern with a SemVP partition and a semantic filter, respectively. In case a match is found, a semantic filter (representing the reduced set of rows) is read to answer a query instead of reading the entire partition for a property. Results demonstrate that SPARTI executes queries around half the time over all query shapes compared to the state-of-the-art while maintaining around an order of magnitude lower space-overhead.

[1]hadoop.apache.org

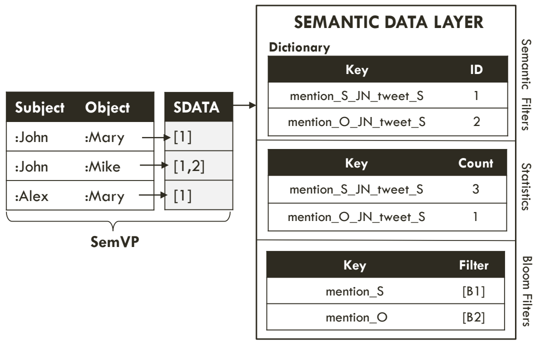

Figure 1: SemVP Structure for a property named :mention where the table consisting of the subject and object columns represents the original triples of triples associated with the :mention property

2 SEMVP

SemVP is a relational partitioning schema that extends vertical partitioning (VP) with row-level semantics. SemVP initially applies vertical partitioning based on the property name [1]. 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 [22], 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.

Figure 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 *semantic filters*. Semantic filters allow SPARTI to determine the rows that contribute to the final result of specific query join patterns. Semantic 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.

In terms of the physical storage, SemVP is best described using a column-store format [2]. Column stores are ideal for handing 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.

2.1 Semantic Data Layer

The semantic data layer consists of three main sections. The first section is a dictionary representing the semantic filters. The semantic filters are defined as key-value pairs, where the key represents the semantic filter identifier and the value represents whether a row qualifies a join pattern or not. Figure 1 lists a set of semantic filters. SDATA associates a semantic filter to a matching row in a partition. For example, the second semantic filter is associated with the second row only. In contrast, the first semantic filter is associated with all rows. The second section represents the statistics for every semantic filter. The statistics indicate the number of rows matching a semantic filter. SPARTI utilizes the statistics to determine the execution order of 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.

2.2 Semantic Filters

Semantic filters consist of two components. The first component is the semantic 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 semantic filter format is described as follows:

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

The semantic 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 semantic filter name denotes the SemVP name.

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

2.2.1 *Join Correlations.*

The BGP in a SPARQL query consists of a set of triples patterns. Joins between the subjects, properties, and objects represent the possible join correlations between triples. Given a pair of 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 [20]. Unbound properties are considered analytic queries and represent a small fraction of SPARQL query-workloads [1]. Similar to other work [1, 2, 20], SPARTI does not optimize queries with unbound properties where it uses the whole RDF dataset to perform such queries.

Consider the following example. The reduced rows for the `:mention` property representing the semantic 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.

The following example shows a SPARQL query with four triple patterns. The BGP joins are as follows:

Example 2:

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

`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 semantic filter names representing all permutations of every BGP join. For example, the semantic filter name `tweet_O_JN_latitude_S_JN_longitude_S` represents one permutation for a join pattern between the three joined properties `:tweet`,`:latitude`, and `:longitude`.

The maximum number of semantic filters for n joined properties is given by:

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

The complete list of semantic 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 semantic 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 semantic filter names.

2.2.2 Computing Semantic Filters. SPARTI employs Bloom-join [8, 16] to compute intermediate join reductions between partitions. Bloom-join determines if an entry in one partition qualifies a join condition with another partition. Bloom-join utilizes a probabilistic data structure termed a Bloom filter [8]. A Bloom filter does not physically store items, but rather hashes 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 representing the join columns filter the rows in both partitions involved in the join and the results are materialized as semantic filters. Figure 2 illustrates an example of a Bloom-join and Bloom filters.

In Figure 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 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 semantic filters. To illustrate, assume that a relation between `:mention` and `:tweet` has been

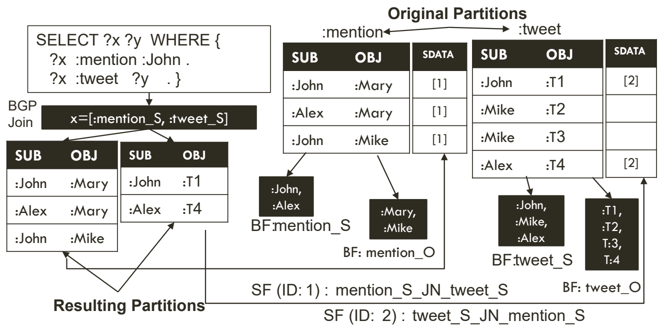

Figure 2: Computing Bloom-join between :mention and :tweet - Bloom Filters (BF), Semantic Filters (SF)

identified. The query consists of a BGP join between `:mention` and `:tweet` on the subject column. SPARTI uses the Bloom filter of `tweet_S` to compute a 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` partition is probed against `tweet_S`. 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 semantic filters (marked as 1 and 2) and are saved in the semantic data layer (i.e., the SDATA layer) of their corresponding SemVP.

3 PROPERTY RELATEDNESS

3.1 Identifying Co-occurrence

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

4 EVOLUTION OVER TIME

The query execution performance *evolves* when semantic filters are created for the frequent join patterns observed in the query-workload. Initially, SPARTI employs VP to answer queries using entire partitions. At that point, SPARTI behaves the same way a VP system would perform (i.e., read the whole partitions corresponding to the query properties). Then, SPARTI monitors queries in order to identify the join patterns that do not invoke any semantic filter.

Specifically, any property of a query that is part of a join pattern and does not invoke a semantic filter has its join pattern marked as candidate for semantic filter creation. SPARTI initiates the partition reduction phase every k queries, where k is a user-specified parameter (e.g., every 100 queries). Finally, the created semantic filters are used whenever a computed semantic filter matches a join pattern in a query.

4.1 Selecting Semantic Filters

Equation 1 illustrates the total number of semantic filters proposed for a given set of properties. There is cost associated with creating and computing semantic 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 semantic filters is derived from the query-workload and the SemVP-based statistics. SPARTI adopts the following utility function to measure the importance of a semantic filter:

$$Utility = \alpha(S) + \beta(R) - \delta(P) \tag{2}$$

Equation 2 introduces three parameters, namely S, R, P [2]. 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 semantic filter belongs to (i.e., the first property in the semantic filter name). Bigger partitions are more important to reduce than smaller ones. Parameter P represents the number of properties that the semantic filter has. A semantic 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 semantic filter. The parameters α, β, and δ are smoothing parameters [21], where $\alpha + \beta + \delta = 1$ with $0 < \alpha, \beta, \delta < 1$.

4.2 Eliminating Semantic Filters

Semantic filters incur a minimal storage overhead. Each semantic 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 a criteria for deleting semantic filters. SPARTI uses Equation 3 to calculate when to delete a semantic filter.

$$T_{exp} = T_0 + [\delta(1 + (k-1)(1 - e^{-\lambda(Utility)})] \tag{3}$$

where T_{exp} is the expiration time of a semantic filter, T_0 is the most recent time-stamp a join pattern has been observed, δ is the default expiration time of a semantic filter (e.g., 1 week), k is the maximum life-span of a semantic filter (e.g., 1 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 semantic 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.

[2]Parameters S, R, P are normalized between 0 and 1 based on every workload snapshot values

5 EXPERIMENTS

SPARTI [3] is compared to S2RDF [20]; a state-of-the-art Spark-based RDF system that runs over Spark. Previously, S2RDF has outperformed H2RDF+, Sempala, PigSPARQL, SHARD, and Virtuoso [20] where it has achieved on average the best query execution performance. S2RDF is built on top of Apache Spark [23]; 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 [4] columnar-store format.

5.1 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 is WatDiv diversified stress testing [3].

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 [6] is the real dataset used. It contains 245 million triples. YAGO2s benchmark queries are used to compare the query execution time [7, 17, 20]. The experiments are conducted using an HP DL360G9 cluster with Intel Xeon E5-2660 realized over 5 nodes. Each node consists of 32GB of RAM, and 52 cores.

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 workload performance, and (6) the query execution performance.

5.2 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.

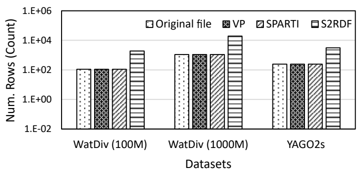

Figure 3: Number of rows across all partitions - Lower is better

5.2.1 Storage Requirements. Figure 3 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.

[3]https://github.com/amgadmadkour/sparti
[4]parquet.apache.org

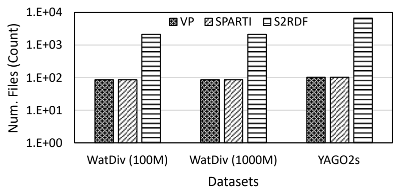

Figure 4: Number of files across all partitions - Lower is better

Figure 4 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 [15].

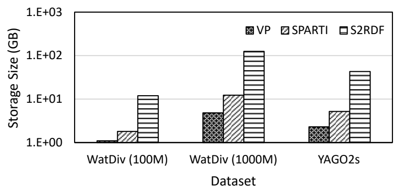

Figure 5: Storage size - Lower is better

Figure 5 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 semantic filters created per partition. S2RDF introduces the highest overhead due to the computed reductions in addition to the original data.

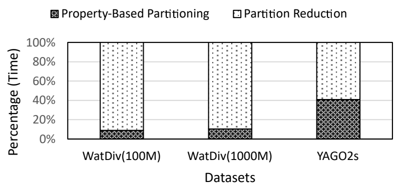

Figure 6: SPARTI load time (in percentage) based on the operating phase

Figure 6 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 7 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 or magnitude faster. This is attributed to the efficiency of Bloom-join reductions versus semi-join reductions employed by S2RDF.

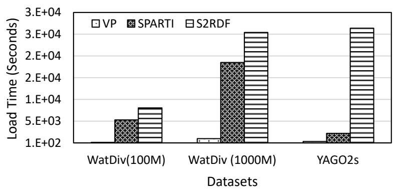

Figure 7: Load time (Seconds) - Lower is better

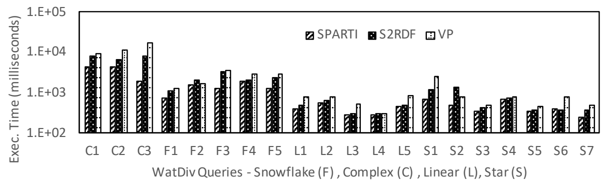

Figure 8: Query Execution Performance (Milliseconds) over WatDiv(1B) - Lower is better

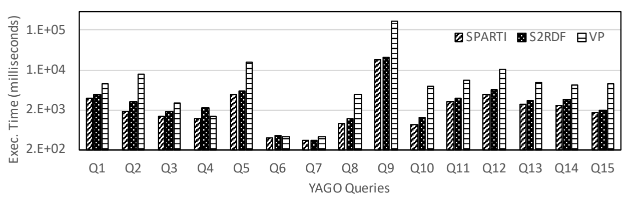

Figure 9: Query Execution Performance (Milliseconds) over YAGO2s - Lower is better

5.2.2 Query Processing Performance. Figures 8 and 9 give the query execution performance of all three systems. Every query is executed five times and the average time is reported. In Figure 8, 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, 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 9 illustrates how SPARTI exhibits better query execution time than the other two systems.

5.2.3 Workload. The WatDiv benchmark provides a stress-testing workload [3] 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.

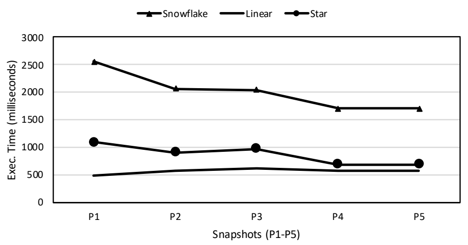

Figure 10: SPARTI query execution performance using WatDiv stress-testing workload for different query shapes. P1-P5 are incremental snapshots of the workload.

Figure 10 gives the query execution performance for the WatDiv queries across five time snapshots of the stress-test workload. The three most common shapes in the work-load (i.e., snowflake, linear, star) exhibit enhanced performance across the five snapshots. This illustrates how SPARTI query performance evolves over time for different query workload shapes.

6 RELATED WORK

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 [14], and Spark-SQL [4] 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 [9] that proposes the creation of an entity-oriented schema from the RDF data. However, performance in DB2RDF is biased towards one query shape. Sempala [19] uses Impala to process SPARQL queries. Sempala stores RDF data as one large property table [22]. This allows star-shaped queries to be answered efficiently as answering queries requires no joins. The aforementioned proposals share one common pattern, namely having a predefined schema. S2RDF [20] proposes an extension to VP, namely ExtVP, where reduced sets of entries are computed for every vertical partition. S2RDF utilizes *semi-join reductions* [5] to reduce the number of rows in a partition. The reduced sets represent all RDF query combinations that appear in SPARQL queries (i.e., Subject-Subject, Subject-Object, Object-Subject, Object-Object). However, S2RDF exhibits a substantial preprocessing overhead that is performed only once. Semi-joins are expensive to compute and generate large network-traffic. S2RDF generates a large number of files to represent the reductions of the original data. SPARTI is similar to S2RDF with respect to employing reductions to achieve better query execution performance. Many techniques utilize the *query-workload* to enhance the partitioning and replication, e.g., WARP [13], PARTOUT [11], and DREAM [12].

7 CONCLUSION

This paper investigates SPARTI, a scalable RDF data management system. SPARTI utilizes a relational scheme that provides row-level semantics for RDF data. The row-level semantics, represented as semantic 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 is used to compute reduced partitions between a set of related properties. The cost-model for managing semantic filters prioritizes the creation of the important semantic filters to compute. Finally, the experimental study that compares SPARTI with the state-of-the-art Spark-based RDF system demonstrates that SPARTI achieves robust performance over synthetic and real datasets.

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Kate Hollenbach. 2007. Scalable Semantic Web Data Management Using Vertical Partitioning. *VLDB* (2007).
[2] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. 2009. SW-Store: A vertically partitioned DBMS for semantic web data management. *VLDB* 18, 2 (2009), 385–406.
[3] Gunes Aluc, Olaf Hartig, M. Tamer Ozsu, and Khuzaima Daudjee. 2014. Diversified Stress Testing of RDF Data Management Systems. *ISWC* (2014), 197–212.
[4] Michael Armbrust, Ali Ghodsi, Matei Zaharia, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, and Michael J. Franklin. 2015. Spark SQL: Relational Data Processing in Spark. *SIGMOD* (2015), 1383–1394.
[5] Philip a. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (1981), 25–40.
[6] Joanna Biega, Erdal Kuzey, and Fabian M Suchanek. 2013. Inside YAGO2s: A transparent information extraction architecture. In *WWW*. 325–328.
[7] Robert Binna, Wolfgang Gassler, Eva Zangerle, Dominic Pacher, and G??nther Specht. 2011. SpiderStore: A native main memory approach for graph storage. In *CEUR*, Vol. 733. 91–96.
[8] B. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[9] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an Efficient RDF Store Over a Relational Database. In *SIGMOD*. 121–132.
[10] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. 2014. Knowledge vault: a web-scale approach to probabilistic knowledge fusion. *KDD* (2014), 601–610.
[11] Luis Galárraga and Ralf Schenkel. 2014. Partout : A Distributed Engine for Efficient RDF Processing. *WWW* (2014), 267–268.
[12] Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, SeyedMehdiReza, Beheshti, and Sherif Sakr. 2015. DREAM : Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *VLDB* (2015), 654–665.
[13] Katja Hose and Ralf Schenkel. 2013. WARP: Workload-aware replication and partitioning for RDF. In *ICDE*. 1–6.
[14] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. *CIDR* (2015).
[15] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, and Christian Bizer. 2015. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.
[16] Lothar F. Mackert and Guy M. Lohman. 1986. R* optimizer validation and performance evaluation for local queries. *SIGMOD Record* 15, 2 (1986), 84–95.
[17] Thomas Neumann and Gerhard Weikum. 2009. The RDF-3X engine for scalable management of RDF data. *VLDB* 19, 1 (2009), 91–113.
[18] Laurens Rietveld, Rinke Hoekstra, and Stefan Schlobach. 2014. Structural Properties as Proxy for Semantic Relevance in RDF Graph Sampling. *ISWC* 8797 (2014), 81–96.
[19] Alexander Schätzle, Martin Przyjaciel-zablocki, Antony Neu, and Georg Lausen. 2014. Sempala: Interactive SPARQL Query Processing on Hadoop. *ISWC* (2014), 164–179.
[20] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF Querying with SPARQL on Spark. *VLDB* 9 (2016), 804—-815.
[21] J.S. Simonoff. 1996. *Smoothing Methods in Statistics*. Springer.
[22] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. 2003. Efficient RDF storage and retrieval in Jena2. *SWDB* (2003), 35–43.
[23] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, and Ankur Dave. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI* (2012), 2–2.