S2RDF: RDF Querying with SPARQL on Spark

Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, Georg Lausen
University of Freiburg, Department of Computer Science
79110 Freiburg im Breisgau, Germany
{schaetzle, zablocki, skilevis, lausen}
@informatik.uni-freiburg.de

ABSTRACT

RDF has become very popular for semantic data publishing due to its flexible and universal graph-like data model. Thus, the ever-increasing size of RDF data collections raises the need for scalable distributed approaches. We endorse the usage of existing infrastructures for Big Data processing like Hadoop for this purpose. Yet, SPARQL query performance is a major challenge as Hadoop is not intentionally designed for RDF processing. Existing approaches often favor certain query pattern shapes while performance drops significantly for other shapes. In this paper, we introduce a novel relational partitioning schema for RDF data called ExtVP that uses a semi-join based preprocessing, akin to the concept of Join Indices in relational databases, to efficiently minimize query input size regardless of its pattern shape and diameter. Our prototype system S2RDF is built on top of Spark and uses SQL to execute SPARQL queries over ExtVP. We demonstrate its superior performance in comparison to state of the art SPARQL-on-Hadoop approaches.

1. INTRODUCTION

RDF is the W3C standard for semantic data modeling. It has a very flexible graph-like data model and thus can be used to represent a large variety from highly to loosely structured datasets. Nowadays, RDF data collections with billions of triples are not unusual, e.g. *Google Knowledge Vault*, raising the need for scalable distributed solutions. One possible approach is to build a standalone distributed RDF store designed primarily for RDF with its own boxed data store, e.g. YARS2 [15]. But this means that data stored in these systems can only be accessed via application specific interfaces or endpoints which hampers interoperability with other systems and causes high integration costs.

On the other side, there already exist mature platforms for distributed *Big Data* processing which are also offered on a rental basis by leading *Cloud* providers, e.g. *Amazon EC2*. *Hadoop* has become one of the de facto industry standards in this area. The key concept is to have a unified pool for data storage (HDFS for Hadoop) that is shared among various applications on top. Thus, different systems can access the same data without duplication or movement for various purposes (e.g. querying, data mining or machine learning). In our view, these existing infrastructures are superior to a specialized deployment in terms of cost-benefit ratio and can provide more synergy benefits. However, as Hadoop is not designed for RDF data management, the main challenge is to achieve performance in the same order of magnitude compared to specialized systems built from ground for RDF.

There exists a lot of work on RDF/SPARQL querying based on *MapReduce* as the execution layer, e.g. [20, 26, 27]. More recently, the emergence of NoSQL key-value stores (e.g. *HBase*, *Accumulo*) as well as in-memory frameworks (e.g. *Impala*, *Spark*) for Hadoop facilitates the development of new systems that are applicable for more interactive workloads, e.g. [23, 28]. Yet still, these systems are typically optimized for query patterns with small diameter like star shapes and small chains. The performance often drops significantly for unselective patterns or queries with long chains.

In this paper, we introduce S2RDF (SPARQL on Spark for RDF), a SPARQL processor based on the in-memory cluster computing framework *Spark*. It comes with a novel partitioning schema for RDF called *ExtVP* (Extended Vertical Partitioning) based on *semi-join reductions* [4] that is an extension of the well-known *Vertical Partitioning* (VP) [1] and is conceptually related to *Join Indices* [30]. In contrast to existing layouts, the optimizations of ExtVP are applicable for all query shapes regardless of its diameter.

Our major contributions can be summarized as follows: (1) We define a novel relational partitioning schema for RDF data called ExtVP that can significantly reduce the input size of a query. (2) As an optional storage optimization, ExtVP allows to define a selectivity threshold to effectively reduce the size overhead compared to VP while preserving most of its performance benefit. (3) We further provide a query compiler from SPARQL to Spark SQL based on ExtVP that uses table statistics to select those tables with the highest selectivity. Our prototype called S2RDF is available for download[1]. (4) Finally, we present a comprehensive evaluation comparing S2RDF with other state of the art SPARQL processors for Hadoop to demonstrate its superior performance on very diverse query workloads. The evaluation is based on the recent synthetic WatDiv [2] benchmark and additionally we use the real-world YAGO [16] dataset to study the effects of the crucial selectivity threshold on query performance and storage overhead.

[1] `http://dbis.informatik.uni-freiburg.de/S2RDF`

Proceedings of the VLDB Endowment, Vol. 9, No. 10

2. RDF & SPARQL

RDF is the W3C recommended standard model for representing information about arbitrary resources. Global identifiers (*IRIs*) are used to identify a resource. For the sake of brevity, we use a simplified notation of RDF without IRIs in the following. The basic notion of data modeling in RDF is a so-called *triple* $t = (s, p, o)$ where s is called *subject*, p *predicate* and o *object*. It models the statement "s has property p with value o" and can be interpreted as an edge from s to o labeled with p, $s \xrightarrow{p} o$. Hence, a set of triples forms a directed labeled (not necessarily connected) graph $G = \{t_1, \ldots, t_n\}$. For example, Figure 1 visualizes RDF graph $G_1 = \{(A, follows, B), (B, follows, C), (B, follows, D), (C, follows, D), (A, likes, I_1), (A, likes, I_2), (C, likes, I_2)\}$.

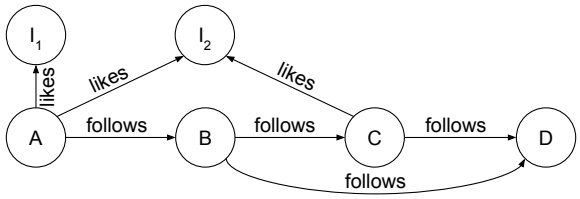

Figure 1: Visualization of RDF graph G_1

SPARQL is the W3C recommended query language for RDF. A SPARQL query Q defines a graph pattern P that is matched against an RDF graph G. This is done by replacing the variables in P with elements of G such that the resulting graph is contained in G (pattern matching). The most basic notion in SPARQL is a so-called *triple pattern* $tp = (s', p', o')$ with $s' \in \{s, ?s\}$, $p' \in \{p, ?p\}$ and $o' \in \{o, ?o\}$, i.e. a triple where every part is either an RDF term (called *bound*) or a variable (indicated by ? and called *unbound*). A set of triple patterns forms a *basic graph pattern* (BGP). For example, the following query Q_1 (in SPARQL syntax) contains a single BGP:

```
SELECT * WHERE {
  ?x likes ?w . ?x follows ?y .                (Q1)
  ?y follows ?z . ?z likes ?w }
```

It can be interpreted as *"For all users, determine the friends of their friends who like the same things"*. Matched on G_1 it gives a single result $(?x \to A, ?y \to B, ?z \to C, ?w \to I_2)$. We use RDF graph G_1 and SPARQL query Q_1 as a running example throughout this paper.

The result of a BGP is a bag of *solution mappings* similar to relational tuples and can be defined analogous to [25]: Let V be the infinite set of query variables and T be the set of valid RDF terms. A *(solution) mapping* μ is a partial function $\mu : V \to T$. We call $vars(tp)$ the set of variables contained in triple pattern tp. Abusing notation, we write $\mu(tp)$ to denote the triple that is obtained by substituting the variables in tp according to μ. The *domain* of μ, $dom(\mu)$, is the subset of V where μ is defined.

Two mappings μ_1, μ_2 are called *compatible*, $\mu_1 \sim \mu_2$, iff for every variable $?v \in dom(\mu_1) \cap dom(\mu_2)$ it holds that $\mu_1(?v) = \mu_2(?v)$. It follows that mappings with disjoint domains are always compatible and the set-union (merge) of two compatible mappings, $\mu_1 \cup \mu_2$, is also a mapping. The answer to a triple pattern tp for an RDF graph G is a bag of mappings $\Omega_{tp} = \{\mu \mid dom(\mu) = vars(tp), \mu(tp) \in G\}$.

The merge of two bags of mappings, $\Omega_1 \bowtie \Omega_2$, is defined as the merge of all compatible mappings in Ω_1 and Ω_2, i.e. $\Omega_1 \bowtie \Omega_2 = \{(\mu_1 \cup \mu_2) \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\}$. It can also be interpreted as a join on the variables that occur in both mappings. Finally, the result to a basic graph pattern $bgp = \{tp_1, \ldots, tp_m\}$ is defined as the merge of all mappings from all triple patterns, $\Omega_{bgp} = \Omega_{tp_1} \bowtie \ldots \bowtie \Omega_{tp_m}$.

On top of these basic patterns, SPARQL also provides more relational-style operators like OPTIONAL and FILTER to further process and combine the resulting mappings. Consequently, the most important aspect to query RDF data efficiently, is an efficient evaluation of BGPs. A formal definition of the SPARQL semantics can also be found in [25].

BGPs in a SPARQL query can have different shapes depending on the position of variables in triple patterns which can have severe impacts on the query performance [2]. The *diameter* of a SPARQL BGP is defined as the longest path, i.e. longest connected sequence of triple patterns, ignoring edge direction. *Star-shaped* patterns have a diameter of one and occur frequently in SPARQL queries, thus many query processors are optimized for this kind of workload. They are characterized by *subject-subject* joins between triple patterns as the join variable is on subject position. *Linear-* or *path-shaped* patterns are also very common in graph querying, e.g. famous friend-of-a-friend queries. Linear patterns are made of *object-subject* (or *subject-object*) joins, i.e. the join variable is on subject position in one triple pattern and on object position in the other. Thus, the diameter corresponds to the number of triple patterns. The performance of such workloads is often worse compared to star-shaped patterns in many RDF triplestores as the selectivity is typically lower and the result sets can become large for highly connected graphs. *Snowflake-shaped* patterns are combinations of several star shapes connected by typically short paths. More complex query structures are essentially compositions of these fundamental patterns.

3. RELATED WORK

In recent years, a large variety of RDF stores have been developed. A comprehensive listing is out of scope for this work, thus we refer the interested reader to more detailed recent surveys [10, 18]. In the following, we describe the work that is most closely related to our work and in particular those which are used in our evaluation (cf. Section 6).

RDF stores can be broadly categorized into *centralized* and *distributed* systems, running on single machine or on a computing cluster (mostly using shared nothing architecture), respectively. Additionally, they can be distinguished by their storage subsystem, i.e. whether they use a relational database to store RDF data (*relational-backed*), non-relational back-ends like key-value stores (*NoSQL-backed*) or deploy an own storage subsystem tailored to RDF (*native*).

Most of the early centralized RDF systems used a relational back-end to materialize RDF data, e.g. *Jena* [7]. But also more recent state of the art systems like Virtuoso [9], *DB2RDF* (RDF support in DB2) [6] and *MonetDB/RDF* [24] use a DBMS back-end. There also exists a bunch of centralized RDF systems that deploy their own RDF tailored storage solutions, e.g. [21, 32]. Most notably, *RDF-3X* [21] creates an exhaustive set of indexes for all RDF triple permutations and aggregated indexes for subsets, resulting in a total of 15 indexes stored in compressed clustered B+ trees. *Hexastore* [32] maintains six indexes for all triple permutations and can be seen as a combination of vertical partitioning [1] and multiple indexing.

3.1 Distributed Systems

From a very general perspective, we can classify existing distributed approaches in three broad categories:

(1) The first type of systems are *standalone* distributed RDF stores, i.e. they are self-contained and primarily dedicated for distributed RDF processing, e.g. [12, 14, 15]. *4store* [14] and *YARS2* [15] are cluster extensions of centralized systems for RDF processing. *TriAD* [12] uses an asynchronous Message Passing protocol for distributed join execution in combination with join-ahead pruning via RDF graph summarization. The METIS graph partitioner is used to partition the input RDF dataset and construct a summary graph. However, graph partitioning is an expensive task and centralized partitioner such as METIS are known to be limited in scalability. Thus, the initial RDF partitioning can become a bottleneck with increasing data size.

(2) The second type of systems use a *federation* of classical centralized RDF stores deployed on all cluster nodes and build a communication and coordination layer on top that distributes the data and (sub)queries, e.g. [11, 13, 17]. They mainly differ in the partitioning strategy used to spread the data across cluster nodes which also impacts the way how queries are split and executed. The general idea is that as much processing as possible is done locally at every node and a global aggregation mechanism merges the partial results.

One of the first approaches was introduced by Huang et al. in [17]. Data gets partitioned (with METIS) such that triples which are nearby in the graph are stored on the same machine. An instance of RDF-3X is used on all cluster nodes to store the allocated partitions and execute (sub)queries. Partition borders can overlap (n-hop guarantee) such that query patterns with a diameter of at most n can be answered locally. However, this imposes an exponential increase in data duplication with increasing n, thus best results are reported for $n = 2$. Performance degrades significantly when queries exceed the n-hop guarantee where MapReduce is used for partial result aggregation.

Partout [11] also uses RDF-3X on every cluster node but partitions RDF data with respect to a query workload such that queries can be processed by a minimum number of nodes in the cluster while maintaining a load balance. However, the typical query workload has to be known in advance and can also lead to suboptimal partitions if the workload changes over time. The concept of *DREAM* [13] differs from other systems in the sense that queries are partitioned instead of data. All nodes in the cluster store a copy of the whole dataset which enables it to completely avoid intermediate data shuffling but only small auxiliary data has to be exchanged. The main drawback of DREAM is that the resources of a single node can become a bottleneck as the whole dataset must be loaded into RDF-3X on every node.

(3) The third type of systems are built on top of existing distributed platforms for *Big Data* processing like *Hadoop*, e.g. [20, 23, 26, 27, 28]. S2RDF falls into this category. Hadoop is also offered on a rental basis by leading *Cloud* providers, e.g. *Amazon Elastic Compute Cloud (EC2)*.

Many of these systems use MapReduce for query execution in some way or another, e.g. [20, 26, 27], as Hadoop originally started as a clone of Google's MapReduce (see [18] for a more comprehensive listing). *SHARD* [26] groups RDF data by subject and uses a so-called Clause-Iteration approach for query processing, i.e. a MapReduce job is created for every triple pattern (called clause) which conceptually leads to a left-deep query plan. *PigSPARQL* [27] uses a vertical partitioning schema for data representation. Instead of compiling SPARQL queries directly into MapReduce jobs, it uses *Pig* as an intermediate layer. By means of this two-level abstraction, PigSPARQL profits from sophisticated optimizations of Pig and runs on all platforms supported by Pig, including all versions of Hadoop. *RAPID+* [20] follows a very similar approach by extending Pig with a so-called *Nested Triple Group Algebra* to reduce the number of MapReduce cycles during query processing. Yet, these systems suffer from relatively high query latencies due to the batch oriented nature of MapReduce. Thus, they are more suited for long running ETL and analytical query workloads.

H2RDF+ [23] is based on *HBase*, a variant of Google's BigTable. HBase is a sorted and column-oriented NoSQL key-value store on top of HDFS. H2RDF+ uses six tables for all possible triple permutations and triples are completely stored in the row key, thus it creates six clustered indexes. Additionally, it also maintains aggregated index statistics to estimate triple pattern selectivity as well as join output size and cost. Based on these estimations, H2RDF+ adaptively decides if queries are executed centralized over a single node or distributed via MapReduce. It comes with implementations of merge and sort-merge joins for both MapReduce and local execution. However, distributed query execution can be orders of magnitude slower than centralized.

Sempala [28] is conceptually related to S2RDF as it is also a SPARQL-over-SQL approach based on Hadoop. It is built on top of *Impala*, a massive parallel processing (MPP) SQL query engine. Its data layout consists of a single unified property table such that star-shaped queries can be answered without joins. Hence, its layout is targeted towards a specific query shape.

4. EXTENDED VERTICAL PARTITIONING

Data layout plays an important role for efficient SPARQL query evaluation in a distributed environment. The most straight forward representation of RDF in a relational model is a so-called *triples table* with three columns, containing one row for each RDF triple, i.e. $TT(s, p, o)$. For efficiency reasons, it must be accompanied by several indexes over some or all (six) triple permutations as query evaluation essentially boils down to a series of self-joins on this large table, e.g. [21, 32]. However, rich indexes are hardly supported by most Hadoop frameworks. An often used optimization is *Vertical Partitioning* (VP), introduced by Abadi et al. in [1]. Instead of a single three-column table, it uses a two-column table for every RDF predicate, e.g. $follows(s, o)$. It mimics the effect of an index on predicates and is also easy to manage in a distributed Hadoop environment.

Regarding the efficient evaluation of SPARQL BGPs in a Hadoop setting, one can conceptually distinguish two design goals: (1) the minimization of input data size and thus I/O in general, and (2) the reduction of join operations. Furthermore, one has also to consider the specific properties of the underlying execution layer, in our case Spark. We therefore examined the main influence factors of query performance in Spark SQL. The most important finding was that reduction of data input size tends to be more effective than reduction of join operations. We attribute this to the fact that Spark is an in-memory system optimized for pipelined execution with little setup overhead for individual operations.

4.1 ExtVP Definition

Many existing data layouts for RDF are tailored towards a specific kind of query shape (cf. Section 2) and most often star-shaped queries are the primary focus as these shapes occur very often in typical SPARQL queries. The goal of our data layout in S2RDF is not to focus on a specific query shape but to provide improvements for all shapes and also for queries with a large diameter, a query type that is often neglected by existing approaches.

Based on our pre-evaluation findings, we decided to use a vertical partitioned (VP) schema as the base data layout for RDF in S2RDF. Using such a schema, the results for a triple pattern with bound predicate can be retrieved by only accessing the corresponding VP table which leads to a large reduction of the input size, in general. Unfortunately, the size of these tables is highly skewed in a typical RDF dataset with some tables containing only a few entries while others comprise a large portion of the entire graph. Hence, there are still a lot of *dangling* tuples, i.e. input tuples that do not contribute to the output of a query, that are potentially shuffled during query execution. They cause unnecessary I/O and comparisons during join execution as well as an increase in memory consumption. Since Spark is an in-memory system and memory is typically much more limited than HDFS disk space, saving this resource is important for scalability. Therefore, to avoid dangling tuples in the query input to a large extent, we developed an extension to the VP schema called *Extended Vertical Partitioning* (ExtVP).

The basic idea is to determine the subsets of a VP table VP_{p_1} that are guaranteed to find at least one match when joined with another VP table VP_{p_2}, p_1, p_2 predicates in G. That is, we precompute a number of semi-join reductions [4] of VP_{p_1}. The relevant semi-joins between tables in VP are determined by the possible joins that can occur when combining the results of triple patterns during query execution. The position of a variable that occurs in both triple patterns (called join variable) determines the columns on which the corresponding VP tables must be joined. We call the co-occurrence of a variable in two triple patterns a *correlation*. Figure 2 illustrates the possible correlations, e.g. if the join variable is on subject position in both triple patterns we call this a *subject-subject correlation* (SS) as both VP tables (here *follows* and *likes*) must be joined on subjects (s). The other correlations are *subject-object* (SO), *object-subject* (OS) and *object-object* (OO). We do not consider join variables on predicate position as such patterns are primarily used for inference or schema exploration but rarely used in a typical SPARQL query [1]. S2RDF can answer such queries by accessing the base triples table for triple patterns with unbound predicate but is not further optimized for it.

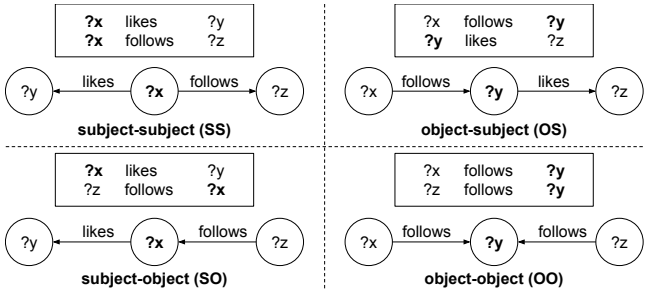

Figure 2: Correlations between triple patterns

We precompute semi-join reductions for SS, OS and SO correlations between all pairs of VP tables (i.e. all pairs of predicates). We omit OO correlations as this would not help much in practice. Two triple patterns in a SPARQL query that are connected by an OO correlation often use the same predicates (cf. OO in Figure 2) and thus result in a self-join of the corresponding VP table. In this case, a semi-join would reduce nothing but simply return the table itself. So we decided not to precompute OO correlations at all due to their relatively poor cost-benefit ratio. Indeed, it is only a design choice and we could precompute them just as well. The advantage of ExtVP is that none of these precomputations are mandatory. S2RDF makes use of ExtVP tables, if they exist, or uses the normal VP tables instead. Furthermore, an optional *selectivity threshold* for ExtVP can be specified such that only those ExtVP tables are materialized where the reduction of original VP tables is large enough. This reduces the size overhead to a large extent as we discuss in more detail in Section 4.3.

To summarize, for two VP tables VP_{p_1}, VP_{p_2} we compute the following semi-join reductions and materialize the results as separate tables in HDFS (if not empty and selectivity is within the threshold):

$$\begin{array}{ll} \textbf{SS:} & VP_{p_1} \ltimes_{s=s} VP_{p_2} \ , \ VP_{p_2} \ltimes_{s=s} VP_{p_1} \\ \textbf{OS:} & VP_{p_1} \ltimes_{o=s} VP_{p_2} \ , \ VP_{p_2} \ltimes_{o=s} VP_{p_1} \\ \textbf{SO:} & VP_{p_1} \ltimes_{s=o} VP_{p_2} \ , \ VP_{p_2} \ltimes_{s=o} VP_{p_1} \end{array}$$

Essentially, the idea of ExtVP comes from the fact that a join between two tables T_1, T_2 on attributes A, B can be decomposed in the following way:

$$T_1 \bowtie_{A=B} T_2 = (T_1 \ltimes_{A=B} T_2) \bowtie_{A=B} (T_1 \rtimes_{A=B} T_2) \qquad (*)$$

This is a common join optimization technique in distributed database systems to reduce overall communication costs [22]. Let $\mathcal{P}$ denote the set of all predicates in an RDF graph G. Formally, an ExtVP schema over G can be defined as:

$$\begin{aligned} ExtVP^{SS}_{p_1|p_2}[G] &= \{(s,o) \mid (s,o) \in VP_{p_1}[G] \wedge \\ &\qquad \exists (s',o') \in VP_{p_2}[G] : s = s'\} \\ &\equiv VP_{p_1}[G] \ltimes_{s=s} VP_{p_2}[G] \\ ExtVP^{SS}[G] &= \{ExtVP^{SS}_{p_1|p_2}[G] \mid p_1, p_2 \in \mathcal{P} \wedge p_1 \neq p_2\} \\ ExtVP^{OS}_{p_1|p_2}[G] &= \{(s,o) \mid (s,o) \in VP_{p_1}[G] \wedge \\ &\qquad \exists (s',o') \in VP_{p_2}[G] : o = s'\} \\ &\equiv VP_{p_1}[G] \ltimes_{o=s} VP_{p_2}[G] \\ ExtVP^{OS}[G] &= \{ExtVP^{OS}_{p_1|p_2}[G] \mid p_1, p_2 \in \mathcal{P}\} \\ ExtVP^{SO}_{p_1|p_2}[G] &= \{(s,o) \mid (s,o) \in VP_{p_1}[G] \wedge \\ &\qquad \exists (s',o') \in VP_{p_2}[G] : s = o'\} \\ &\equiv VP_{p_1}[G] \ltimes_{s=o} VP_{p_2}[G] \\ ExtVP^{SO}[G] &= \{ExtVP^{SO}_{p_1|p_2}[G] \mid p_1, p_2 \in \mathcal{P}\} \\ ExtVP[G] &= \{ExtVP^{SS}[G], ExtVP^{OS}[G], ExtVP^{SO}[G]\} \end{aligned}$$

Important to notice is that we do not precompute the actual join results themselves as this would usually increase space consumption by an order of magnitude or even more. Semi-join reductions on the other side are always guaranteed to be a subset of the corresponding base table. Hence, space consumption and effectiveness of ExtVP depends on the selectivity of corresponding semi-joins. In practice, the space requirement of ExtVP is reasonable and comparable to existing approaches, e.g. [23, 28, 32] (cf. Section 4.3).

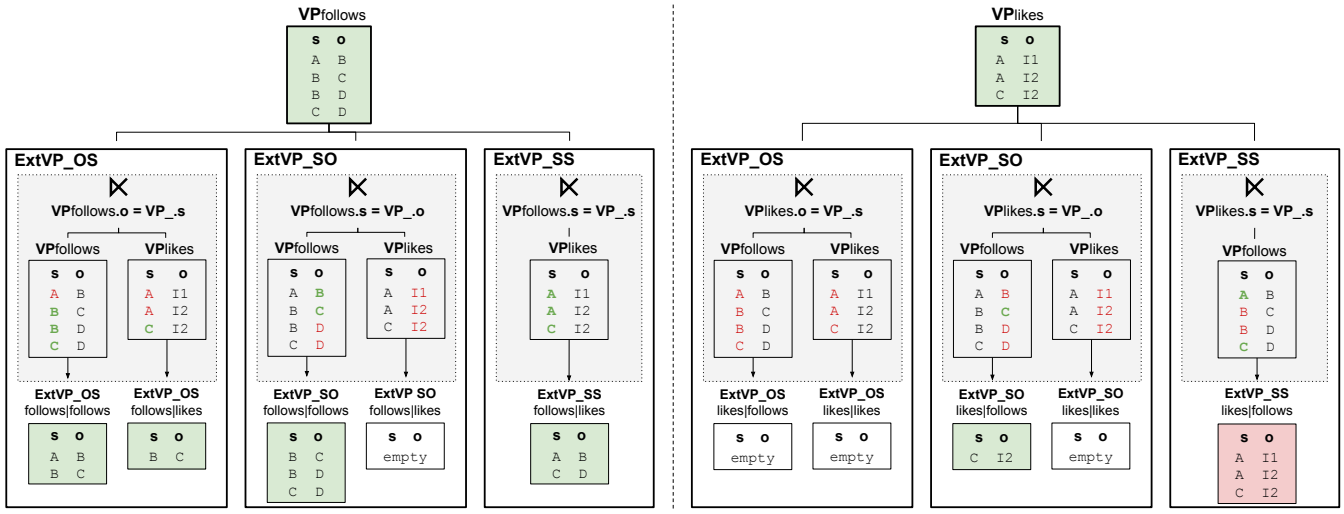

Figure 3: ExtVP data model for RDF graph G_1. Left side shows ExtVP tables derived from $VP_{follows}$, right side for VP_{likes}, resp. Tables marked in green are stored. Red tables are equal to VP and thus not stored.

Figure 3 illustrates the entire ExtVP schema for our running example RDF graph G_1 (cf. Section 2). In a typical heterogeneous RDF dataset consisting of several classes, many ExtVP tables would be empty because their predicates can not be combined (e.g. users are likely described by different predicates than products). To avoid these unnecessary semi-join operations when constructing an ExtVP schema, we determine those predicates that have any correlation to one another and precompute only these reductions. For example, the following query determines those predicates that have an OS correlation to predicate p_1:

```
SELECT DISTINCT TT.p FROM TriplesTable TT
LEFT SEMI JOIN VPp1 ON TT.o = VPp1.s
```

Finally, we discuss the updatability of ExtVP. Insertions are not critical since we can easily adapt the corresponding ExtVP tables, i.e. we can append semi-join results for new triples. Deletions are a bit more affected. To remove a triple (s, p, o) we have to delete corresponding tuples from all ExtVP tables for p which potentially leads to dangling tuples in other tables. Even so, query results are still correct as the actual joins between ExtVP tables are performed on query runtime where potential dangling tuples get discarded (cf. $(*)$). However, they might degrade the quality of optimization provided by ExtVP, to some extent. As usual, updates can be realized by a combination of delete and insert. In our current implementation, deletions can not be realized as HDFS is an immutable append-only filesystem. Thus, to support updates and deletions, S2RDF needs to be extended by an updatable storage middle layer like HBase.

4.2 ExtVP and Database Query Optimization

The notion of semi-joins was originally proposed to reduce communication cost for join processing in early distributed databases [4]. However, in Spark joins are executed in parallel on all cluster nodes on portions of the data, similar to an MPP (Massively Parallel Processing) database. This makes the application of semi-joins on the fly during query processing less effective. In contrast, we precompute semi-join reductions of VP tables for all possible correlations (omitting OO correlations for aforementioned reasons) such that we do not have to compute them on-the-fly but only once in advance. We can afford this because VP tables, in contrast to an arbitrary relational schema, have a fixed two-column layout. Conceptually, the idea of ExtVP is related to the notion of *Join Indices* [30] in relational databases and *Access Support Relations* [19] in object-oriented databases. An $ExtVP_{p_1|p_2}$ table basically resembles a *clustered join index* between VP tables VP_{p_1} and VP_{p_2} as we store the actual payload in the index instead of using unique surrogates. Access support relations (ASR) have been introduced to facilitate path expression evaluation. In principle, $ExtVP^{OS}$ resembles a binary decomposition of all possible ASR (i.e. paths) in an RDF graph following edges in forward direction. Complementary, $ExtVP^{SO}$ can be seen as a binary decomposition following edges in backward direction.

4.3 ExtVP Selectivity Threshold

ExtVP comes at the cost of additional storage overhead compared to VP. But as the effectiveness of ExtVP increases with smaller tables sizes (due to higher selectivity), we can reduce this overhead to a large extent while retaining most of its benefits. Let SF be the *selectivity factor* of a table in ExtVP, i.e. its relative size compared to the corresponding VP table: $SF(ExtVP_{p_1|p_2}) = |ExtVP_{p_1|p_2}|/|VP_{p_1}|$. For example, $ExtVP^{OS}_{follows|likes}$ in Figure 3 has a SF value of 0.25 as its size is only a quarter of $VP_{follows}$. Let $k = |\mathcal{P}|$ be the number of predicates in an RDF graph G and $n = |G|$ be the number of triples in G. It holds that the sum of all tuples in VP, $|VP[G]|$, is also equal to n. W.l.o.g. we assume that all VP tables have equal size n/k and $SF = 0.5$ for all ExtVP tables. The size of an ExtVP schema for G (i.e. sum of all tuples) can then be estimated as follows:

$$|ExtVP[G]| = \underbrace{k}_{\#\text{predicates}} * (\underbrace{(3k-1)}_{\substack{\#\text{tables} \\ \text{per predicate}}} * \underbrace{\frac{n}{2k}}_{\text{table size}})$$
$$= (3k-1) * \tfrac{n}{2} < \tfrac{3}{2}kn$$

This is, however, by far an overestimation of the real size as it assumes that all predicates can be combined with one another. In our experiments, typically more than 90% of all ExtVP tables were either empty or equal to VP and hence

not stored. In general, the more predicates exist in an RDF dataset the more ExtVP tables will be empty as many of these predicates have distinct domains (e.g. predicates describing products vs. users). Exemplary, for a dataset with $n \approx 10^9$ triples and 86 predicates the actual size of ExtVP was $\sim 11n$ (cf. Section 6). HDFS storage space is normally not a limiting factor in a Hadoop environment and as we use the *Parquet* columnar storage format in combination with *snappy* compression to materialize the tables in HDFS, the physical size of ExtVP (including VP tables) was ~ 1.3 times the original input RDF dataset size in N-triples format.

Nonetheless, $11n$ tuples in ExtVP compared to n tuples in VP states a significant overhead. On the one hand, tables with $SF \sim 1$ impose a large overhead while contributing only a negligible performance benefit. On the other hand, tables with $SF < 0.25$ give the best performance benefit while causing only little overhead. To this end, S2RDF supports the definition of a *threshold* for SF such that all ExtVP tables above this threshold are not considered. As demonstrated in Section 6.3, a threshold of 0.25 reduces the size of ExtVP from $\sim 11n$ to $\sim 2n$ tuples and at the same time provides 95% of the performance benefit on average compared to using no threshold.

5. S2RDF QUERY PROCESSING

Spark [33] is a general-purpose in-memory cluster computing system that runs on Hadoop and can process data from any Hadoop data source and Spark SQL [3] is the relational interface of Spark. We use the general-purpose *Parquet* columnar storage format to persist the data store of S2RDF in HDFS. Parquet is not Spark exclusive and thus we could directly load and query the data with *Impala* just as well without any need for data movement or preparation.

Query processing in S2RDF is based on the algebraic representation of SPARQL expressions. We use *Jena ARQ* [7] to parse a SPARQL query into the corresponding algebra tree and apply some basic algebraic optimizations, e.g. filter pushing. SPARQL query optimization was not a core aspect when developing S2RDF, hence there is still much room for improvement in this field. Finally, the tree is traversed from bottom up to generate the equivalent Spark SQL expressions based on our ExtVP schema described in Section 4. That is, an input SPARQL query gets mapped to a single equivalent Spark SQL query that is then executed by Spark. We describe the details of this mapping in the following.

5.1 Triple Pattern Mapping

The basic concept is that every triple pattern in a BGP $bgp = \{tp_1, \ldots, tp_n\}$ is represented by an equivalent subquery $\{sq_1, \ldots, sq_n\}$ and the results of these subqueries are joined to compute the result of bgp, $\Omega_{bgp} = sq_1 \bowtie \ldots \bowtie sq_n$. For this, the query compiler of S2RDF has to select the appropriate table for every triple pattern tp_i. In a VP schema, this choice is unambiguous as it is simply defined by the predicate of tp_i. In an ExtVP schema, however, there are potentially several candidate tables defined by the correlations of tp_i to other triple patterns in bgp. From these candidates, the table with the best selectivity factor SF should be chosen. S2RDF collects statistics about all tables in ExtVP during the initial creation process, most notably the selectivities (SF values) and actual sizes (number of tuples), such that these statistics can be used for query generation. It also stores statistics about empty tables (which do not physically exist) as this empowers the query compiler to know that a query has no results without actually running it.

The table selection procedure is depicted in Algorithm 1. If the predicate of the input triple pattern tp_i is a variable, we have to use the base triples table (or union of VP tables) to match that pattern. If not, it initially starts with the corresponding VP table and iterates over all other triple patterns in the input BGP to check whether they have any correlation (SS, SO, OS) to tp_i. If tp_i has more than one correlation to another triple pattern, the algorithm selects the corresponding ExtVP table with smallest (best) SF value.

Algorithm 1: TableSelection

input: $TriplePattern\ tp_i : (s, p, o)$
$BGP : Set\langle TriplePattern : (s, p, o)\rangle$
output: $tab : Table$

if $isVar(tp_i.p)$ **then return** $TriplesTable$
else $tab \leftarrow VP_{tp_i.p}$ `// initially start with VP table`
foreach $tp : TriplePattern \in BGP \neq tp_i$ **do**
  if $tp_i.s = tp.s \wedge SF(ExtVP^{SS}_{tp_i.p|tp.p}) < SF(tab)$ **then**
    $tab \leftarrow ExtVP^{SS}_{tp_i.p|tp.p}$ `// SS correlation`
  if $tp_i.s = tp.o \wedge SF(ExtVP^{SO}_{tp_i.p|tp.p}) < SF(tab)$ **then**
    $tab \leftarrow ExtVP^{SO}_{tp_i.p|tp.p}$ `// SO correlation`
  if $tp_i.o = tp.s \wedge SF(ExtVP^{OS}_{tp_i.p|tp.p}) < SF(tab)$ **then**
    $tab \leftarrow ExtVP^{OS}_{tp_i.p|tp.p}$ `// OS correlation`
return tab

Exemplary, consider triple pattern $tp_3 = (?y, follows, ?z)$ in Figure 4. It has a SO correlation to $tp_2 = (?x, follows, ?y)$ on variable $?y$ and an OS correlation to $tp_4 = (?z, likes, ?w)$ on variable $?z$. Hence, in total, there are three candidate tables to answer tp_3: (1) $VP_{follows}$, (2) $ExtVP^{SO}_{follows|follows}$ and (3) $ExtVP^{OS}_{follows|likes}$. From these tables, (3) gets selected as it has the best SF value.

Once the appropriate table is selected, the corresponding SQL subquery to retrieve the results for triple pattern tp_i can be derived from the position of variables and bound values in tp_i. Bound values are used as conditions in the WHERE clause and variable names are used to rename table columns in the SELECT clause such that all subqueries can be easily joined on the same column names, i.e. using natural joins. The mapping of a triple pattern to SQL is depicted in Algorithm 2 using relational algebra notation. It checks all positions of tp_i whether they contain a variable or bound value. In case of a variable, the corresponding column gets renamed by the variable name and added to a list of projections (we combine rename and projection for shorthand notation). For a bound value on subject or object position, the corresponding selection is added to a list of conditions. A bound predicate is already covered by the selected table and thus no additional selection is needed. In Figure 4 the corresponding SQL subqueries for triple patterns $tp_1, \ldots, tp_4$ are given on the right.

5.2 Query Composition

To compute the overall BGP result, the subqueries for all triple patterns must be joined. Regarding query semantics, the order of triple patterns in a SPARQL BGP does not affect the query result. However, when a query is evaluated, the order in which triple patterns and thus subqueries are actually executed can have severe impacts on performance.

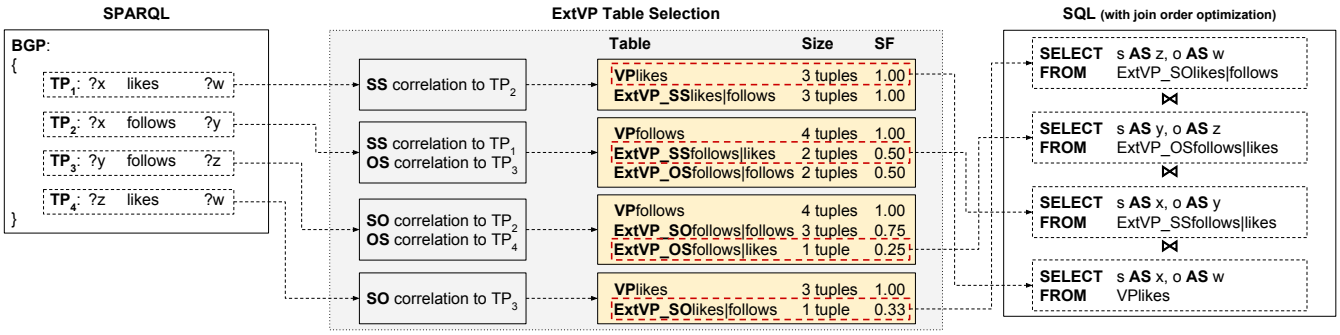

Figure 4: SPARQL to SQL for Q_1 based on ExtVP. Correlations between triple patterns determine the possible ExtVP tables. From these candidate tables, the ones with best (min) SF values get selected.

Algorithm 2: TP2SQL

input: $TriplePattern\ tp_i : (s,p,o)$, $tab : Table$
output: $query$: SQL (in relational algebra notation)

1 $projections \leftarrow \emptyset$, $conditions \leftarrow \emptyset$
2 **if** $isVar(tp_i.s)$ **then**
3 | $projections \leftarrow projections \cup (s \rightarrow tp_i.s)$
4 **else** $conditions \leftarrow conditions \cup (s = tp_i.s)$
5 **if** $isVar(tp_i.p)$ **then**
6 | $projections \leftarrow projections \cup (p \rightarrow tp_i.p)$
7 **if** $isVar(tp_i.o)$ **then**
8 | $projections \leftarrow projections \cup (o \rightarrow tp_i.o)$
9 **else** $conditions \leftarrow conditions \cup (o = tp_i.o)$
10 **return** $query \leftarrow \pi[projections]\sigma[conditions](tab)$

Algorithm 3: BGP2SQL_OPT

input: $BGP : Set\langle TriplePattern : (s,p,o)\rangle$
output: $query$: SQL (in relational algebra notation)

1 $vars \leftarrow \emptyset$, $query \leftarrow \emptyset$
2 $tmpBGP \leftarrow orderByBoundValues(BGP)$
3 **while** $tmpBGP \neq \emptyset$ **do**
4 | $tp_{next} \leftarrow \emptyset$, $tab_{next} \leftarrow \emptyset$
5 | **foreach** $tp : TriplePattern \in tmpBGP$ **do**
6 | | $tab \leftarrow$ TableSelection(tp, BGP)
7 | | **if** $tab_{next} = \emptyset \vee$
8 | | $(Size(tab) < Size(tab_{next}) \ \wedge \ vars \cap vars(tp) \neq \emptyset)$
| | **then**
9 | | | $tp_{next} \leftarrow tp$, $tab_{next} \leftarrow tab$
10 | **if** $SF(tab_{next}) = 0$ **then return** $\emptyset$
11 | **else if** $vars = \emptyset$ **then**
12 | | $query \leftarrow$ TP2SQL(tp_{next}, tab_{next})
13 | **else** $query \leftarrow query \bowtie$ TP2SQL(tp_{next}, tab_{next})
14 | $vars \leftarrow vars \cup vars(tp_{next})$
15 | $tmpBGP \leftarrow tmpBGP \setminus \{tp_{next}\}$
16 **return** $query$

In S2RDF, for a BGP with n triple patterns we derive $n-1$ join operations between ExtVP tables in the generated SQL query to compute the result. As query workload is typically I/O bound, it is crucial to optimize the join order such that the amount of intermediate results gets reduced.

Let $sel(tp) = |\Omega_{tp}|/|G|$ be the *selectivity* of a triple pattern tp for RDF graph G. The general rule of thumb is to order triple patterns by selectivity, i.e. $sel(tp_i) < sel(tp_{i+1})$. This is based on the assumption that smaller join inputs usually lead to smaller join outputs which is not always true but tend to be a good approximation in practice. The most obvious optimization is to make sure that patterns with more bound values are executed first and cross joins are avoided as they have the worst selectivity, i.e. it should hold that $(\bigcup_{j \leq i} vars(tp_j)) \cap vars(tp_{i+1}) \neq \emptyset$. This can be derived statically from the query structure itself.

In addition, S2RDF can make use of table statistics collected during initial ExtVP creation. As the system is aware of the size of each VP and ExtVP table, it can order those triple patterns with the same amount of bound values by size of the corresponding table that is selected by Algorithm 1. It uses the actual table sizes instead of selectivity factors as we want to join the smallest tables first and not the ones with the highest reduction compared to VP. This procedure is depicted in Algorithm 3. It first orders triple patterns by the number of bound values (line 2). From this list it iteratively picks the triple pattern with smallest corresponding ExtVP table (lines 5-9) and adds a join operation for this table to the generated SQL query (lines 11-13). This defines the order of joins performed by Spark when executing the final SQL query. If one of the selected tables is empty, i.e. $SF = 0$ (line 10), it can directly return an empty result.

Consider again our running example in Figure 4. All triple patterns have the same amount of bound values and no cross joins occur when processing them in the given order. However, execution in listed order would first join the two largest tables and produce intermediate results that get discarded in the following joins. Instead, as S2RDF is aware of the size of all tables, it places the subqueries corresponding to tp_3 and tp_4 at the beginning of join order as they use the smallest tables (cf. Figure 4). Overall, this reduces the intermediate result size thus saving I/O, and also the total number of join comparisons thus saving CPU.

The remaining SPARQL 1.0 operators can be more or less directly mapped to the appropriate counterparts in Spark SQL. A FILTER expression in SPARQL can be mapped to equivalent conditions in Spark SQL where we essentially have to adapt the SPARQL syntax to the syntax of SQL. These conditions can then be added to the WHERE clause of the corresponding (sub)query. OPTIONAL is realized by a left outer join while UNION, OFFSET, LIMIT, ORDER BY and DISTINCT are realized using their equivalent clauses in the SQL dialect of Spark. S2RDF does currently not support the additional features introduced in SPARQL 1.1, e.g. subqueries and aggregations. This is left for future work.

Eventually, a given SPARQL input query is represented by a single equivalent Spark SQL query which is then hand over to Spark for execution.

6. EVALUATION

Evaluation was performed on a small cluster of 10 machines (1 master and 9 worker), each equipped with an Intel Xeon E5-2420 CPU @1.90GHz, 2x2 TB disks, 32 GB RAM running Ubuntu 14.04 LTS and connected via Gigabit Ethernet. We used the Hadoop distribution of Cloudera CDH 5.4 which was the most recent version at the time of evaluation. For Spark we used the version shipped with CDH (Spark 1.3). Broadcast joins in Spark SQL were disabled since they are based on Hive table statistics which are not available since we do not store ExtVP tables in the Hive Metastore. Consequently, they would be based on inaccurate size estimations. S2RDF uses its own statistics for table selection. In this way, we do not impose an additional dependency on Hive. Each Spark executor was given 20 GB of memory and Parquet filter pushdown was enabled.

We compare S2RDF with state of the art SPARQL processors for Hadoop. SHARD [26] and PigSPARQL [27] are both based on MapReduce, Sempala [28] uses Impala and H2RDF+ [23] is based on HBase. Additionally, we also compare S2RDF to Virtuoso Open Source Edition v7.1.1 [9] installed on single server with an Intel Xeon X5667 CPU @3.07GHz, 12 TB disk in hardware RAID 5 optimized for read and 32 GB RAM running Ubuntu 14.04 LTS. We include Virtuoso to give a rough idea of how the runtimes compare to a state of the art centralized RDF store. In this way, we demonstrate that our approach to adapt a general-purpose Big Data platform is competitive while at the same time being able to *scale out* (more machines) instead of *scale up* (more resources per machine).

The experiments were conducted on two datasets with approx. 100 million and 1 billion RDF triples generated using the WatDiv Data Generator with scale factors 1000 and 10000, respectively. It is provided by the *Waterloo SPARQL Diversity Test Suite* (WatDiv) [2] that covers all different query shapes and thus allows us to test the performance of S2RDF and competitors in a more fine-grained way. We used the WatDiv Query Generator to instantiate query templates and report the average mean runtimes (AM) in the following. When an ExtVP table is used for the first time, S2RDF uses the `cacheTable` functionality of Spark SQL to cache it in memory. We do not include caching times in our reported query runtimes as it is a one-time operation not required for subsequent queries accessing the same table.

The loading times and store sizes are listed in Table 1. For the largest dataset (SF10000), ExtVP consists of 2043 tables. The other potential tables were either empty (19780 with $SF = 0$) or equal to VP (279 with $SF = 1$). S2RDF needs significantly more time to load the data than the other systems due to the large amount of semi-joins. However, this is a one-time task and we have not spent much effort to optimize this process. In a production environment, one could think of a pay as you go approach where ExtVP tables are computed lazily on the fly when required for the first time in a query and materialized for usage in later queries.

Table 1: WatDiv loading times and HDFS sizes for S2RDF VP/ExtVP ($0 < SF < 1$) and competitors

		SF1000	SF10000
tuples	**original**	109.2 M	1091.5 M
	VP	109.2 M	1091.5 M
	ExtVP	1197.9 M	11967 M
tables	**VP**	86	86
	ExtVP	2041	2043
	total	2127	2129
HDFS size	**original**	5.3 GB	54.9 GB
	ExtVP	6.2 GB	63.7 GB
	H2RDF+	5.2 GB	57.0 GB
	Sempala	3.5 GB	40.4 GB
	PigSPARQL	8.9 GB	92.5 GB
	SHARD	9.9 GB	100 GB
loading	**S2RDF VP**	290 s	1065 s
	S2RDF ExtVP	9497 s	60572 s
	H2RDF+	507 s	5425 s
	Sempala	333 s	2782 s
	PigSPARQL	71 s	498 s
	SHARD	134 s	1222 s

6.1 WatDiv Basic Testing

WatDiv comes with a set of 20 predefined query templates called *Basic Testing* use case which can be grouped in four categories according to their shape: *star* (S), *linear* (L), *snowflake* (F) and *complex* (C). Figure 5 compares the different systems on the largest dataset (SF10000), corresponding AM runtimes are listed in Table 2.

We can observe that S2RDF outperforms both SHARD and PigSPARQL by several orders of magnitude for all query categories due to their underlying MapReduce-based batch execution engine. These systems scale very smoothly with the data size but are not able to provide interactive query runtimes as MapReduce imposes a large startup overhead. The performance of PigSPARQL is better than SHARD due to its multi-join optimization that can process several triple patterns in a single MapReduce job whereas SHARD uses one job per triple pattern.

Though Sempala is optimized for star queries, S2RDF outperforms it by up to an order of magnitude for this category as well (S1-S7). We attribute this to the fact that Sempala, though not performing any joins, has to scan through the whole property table to find the matching rows thus query execution is limited by table scanning time. S2RDF, in contrast, can significantly reduce the input size. For example, the star pattern of query S3 contains triple patterns with predicates `rdf:type` and `sorg:caption` and the corresponding table $ExtVP^{SS}_{type|caption}$ has selectivity $SF = 0.02$, i.e. it reduces the input size for triple pattern with predicate `rdf:type` to only 2%. This underpins our finding from Section 4 that input size reduction is often more effective than the reduction of join operations. Sempala also shows a good performance for other query types but is not able to beat S2RDF for any query. On average, S2RDF is an order of magnitude faster than Sempala for all four query categories.

The performance of H2RDF+ strongly depends on its execution model. For the smaller dataset (SF1000), it can use efficient centralized merge joins for most of the queries. However, for the larger dataset (SF10000), many queries become too costly for centralized execution and MapReduce must be used. For example, while both systems have nearly the same performance for query F1 on SF1000, S2RDF outperforms H2RDF+ by two orders of magnitude for the same query on SF10000. The same is true for the majority of queries regardless of their type. There are only a few queries (5 out of 20) where runtimes of H2RDF+ are slightly better than S2RDF on the largest dataset. These queries contain highly selective triple patterns which are ideal candidates for centralized execution based on HBase lookups. On average, S2RDF outperforms H2RDF+ by at least one order of magnitude for all four query categories.

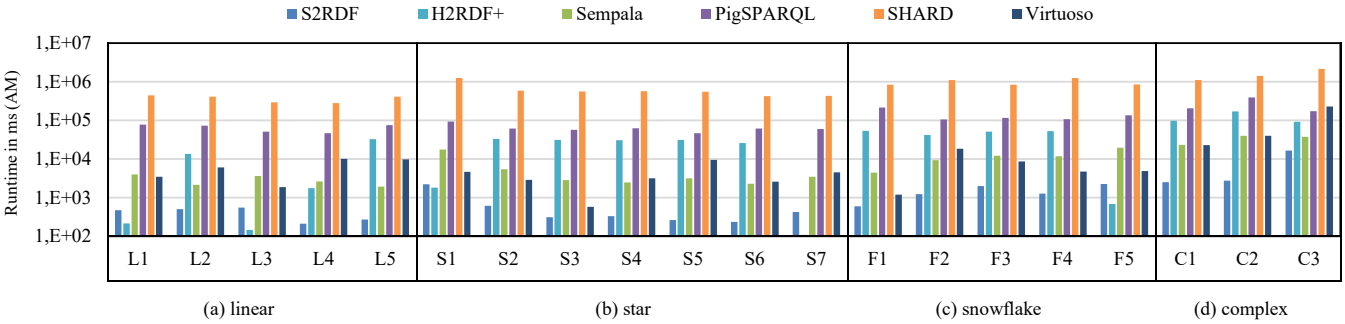

Figure 5: WatDiv Basic Testing (SF10000) of S2RDF and competitors

Table 2: WatDiv Basic (in ms), AM-X = AM of category X, AM-T = total AM of all queries

	Query	L1	L2	L3	L4	L5	AM-L	S1	S2	S3	S4	S5	S6	S7	AM-S
SF1000	S2RDF ExtVP	202	196	196	132	162	178	735	294	219	209	199	209	191	294
	H2RDF+	98	1450	62	207	1616	687	942	3831	2423	1937	996	673	51	1550
	Sempala	904	746	740	664	752	761	3130	1058	862	876	960	848	870	1229
	PigSPARQL	68000	67500	46500	41000	68000	58200	57000	46000	44000	46500	42000	46500	46500	46929
	SHARD	103153	99905	72603	67937	99595	88639	264976	130625	122958	127587	123832	99866	102040	138840
SF10000	S2RDF ExtVP	471	498	549	209	270	399	2208	607	311	329	260	235	420	624
	H2RDF+	214	13447	145	1752	32222	9556	1809	32902	30844	30419	30807	25795	109	21812
	Sempala	3938	2140	3630	2616	1914	2848	17386	5368	2816	2442	3142	2260	3476	5270
	PigSPARQL	76500	72500	51000	46500	75000	64300	92500	61000	56500	62000	46500	61500	59000	62714
	SHARD	444511	406007	288980	275534	406591	364324	1240267	579207	555911	567297	550959	424402	428905	620992
	Virtuoso (cold)	16009	15727	7580	46287	44260	25973	15873	13072	2726	11445	1910	6779	7084	8413
	Virtuoso (AM)	3437	5984	1863	10055	9710	6210	4630	2853	576	3157	9470	2555	4516	3965

	Query	F1	F2	F3	F4	F5	AM-F	C1	C2	C3	AM-C	AM-T
SF1000	S2RDF ExtVP	433	642	638	692	672	615	923	1460	2929	1771	567
	H2RDF+	476	2926	1952	5642	203	2240	53032	20250	35617	36300	6719
	Sempala	1068	1704	1538	1950	2545	1761	2828	5992	6040	4953	1804
	PigSPARQL	97500	80500	92500	85500	99000	91000	119500	272500	64000	152000	76525
	SHARD	182215	237370	181606	266426	181581	209839	236828	297344	298759	277644	164860
SF10000	S2RDF ExtVP	590	1226	1969	1265	2254	1461	2508	2740	16407	7218	1766
	H2RDF+	52843	41480	51085	52323	679	39682	96777	170473	91189	119480	37866
	Sempala	4420	9316	12090	11668	19516	11402	23136	39710	37462	33436	10422
	PigSPARQL	212000	105500	115500	106000	133500	134500	205000	387000	172000	254667	109850
	SHARD	835514	1103997	834778	1245115	841380	972157	1100372	1398989	2146928	1548763	783782
	Virtuoso (cold)	3997	30627	31144	11374	11362	17701	22801	39711	226129	96214	28295
	Virtuoso (AM)	1197	18376	8629	4705	4879	7557	22801	39711	226129	96214	19262

For Virtuoso we observed that queries with empty results are generally very fast due to the use of sophisticated indexes. Furthermore, if a query is executed several times for different template parameter values, its performance improves gradually probably due to caching effects. In Table 2 we report both, the runtimes of Virtuoso for cold caches where we execute every query only once as well as AM runtimes for repeated query execution. Nonetheless, S2RDF is an order of magnitude faster than Virtuoso on average for all four query categories, even with repeated query execution.

Overall, the evaluation clearly demonstrates the superior performance of S2RDF compared to other state of the art distributed and centralized RDF stores for all query shapes. In contrast to existing approaches, ExtVP does not favor any specific query type and achieves consistent performance regardless of query shape. Thus, S2RDF answers most queries in less than a second on a billion triples RDF graph.

6.2 WatDiv Incremental Linear Testing (IL)

The Basic Testing use case is well suited to test the performance for different query shapes but most queries have a rather small diameter. In fact, only two have a diameter larger than 3 (C1, C2). Most RDF stores are optimized for small diameter queries as RDF datasets are typically fragmented based on hashing or graph partitioning algorithms that try to place vertices on the same cluster node that are close to each other in the RDF graph, e.g. [17, 26, 28].

For that reason, we have designed an additional WatDiv use case called *Incremental Linear Testing* focusing on linear queries with increasing size (diameter). It consists of three query types (IL-1, IL-2, IL-3) which are bound by user, retailer or unbound. Each type starts with a diameter of 5 (i.e. 5 triple patterns) and we incrementally add triple patterns (up to 10). For example, query IL-1-8 is a user bound query with diameter 8. This use case is now officially included in WatDiv[2] which emphasizes the importance of such a workload that was not adequately covered before. Figure 6 compares all systems on the largest dataset (SF10000), corresponding AM runtimes are listed in Table 3.

The first query type (IL-1) starts from an instantiated user following various edges along the graph, i.e. it has the structure `( %u% p1 ?v1 . ?v1 p2 ?v2 . ?v2 p3 ?v3 . ... )`. We observe that S2RDF significantly outperforms all other distributed competitors while runtimes rise only slightly with increasing data size. The queries make use of the two largest

[2] `http://dsg.uwaterloo.ca/watdiv/`

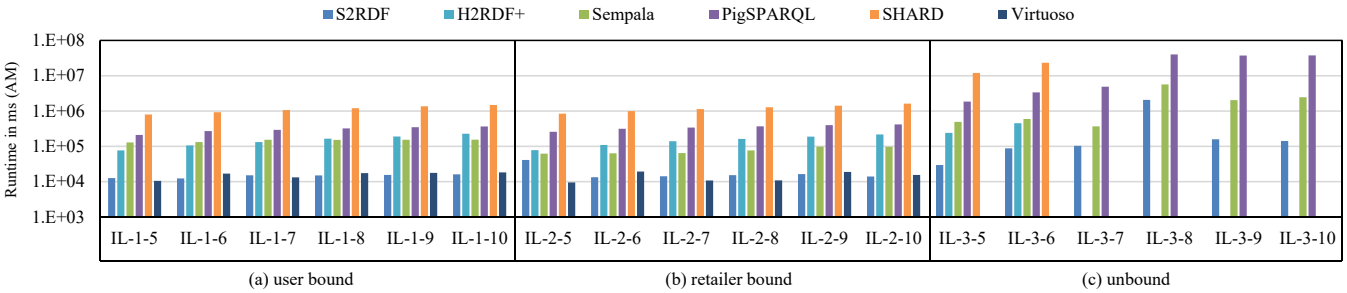

Figure 6: WatDiv Incremental Linear Testing (SF10000) of S2RDF and competitors

Table 3: WatDiv IL (in ms), AM-IL-X = AM of query type X, AM-Y = AM of queries with length Y

	Query	IL-1-5	IL-1-6	IL-1-7	IL-1-8	IL-1-9	IL-1-10	AM-IL-1	IL-2-5	IL-2-6	IL-2-7	IL-2-8	IL-2-9	IL-2-10	AM-IL-2
SF1000	S2RDF ExtVP	1724	1745	1965	2029	2185	2643	2048	4944	1869	1980	2114	2382	2413	2617
	H2RDF+	25351	48957	78117	71448	99636	92232	69290	25090	49745	50514	73750	122180	99416	70116
	Sempala	29321	29684	29595	29696	29658	29663	29603	19357	19388	19496	19867	20162	20152	19737
	PigSPARQL	132130	163757	181044	205225	227679	255845	194280	198302	252621	269446	301519	324526	339143	280926
	SHARD	167910	189782	217487	244575	275592	302084	232905	167761	196993	223118	252725	281614	309606	238636
SF10000	S2RDF ExtVP	12543	12252	15062	15003	15478	16124	14410	41188	13276	14182	15261	16313	13922	19024
	H2RDF+	76284	105794	131672	164583	188800	227637	149128	77567	108780	139282	161913	187288	216887	148620
	Sempala	128486	131304	152730	152169	153360	154272	145387	61843	63501	64487	76717	97933	96590	76845
	PigSPARQL	209594	270757	293241	321021	348274	364243	301188	258307	313681	340580	365995	396331	415046	348323
	SHARD	792204	925542	1064010	1195541	1354956	1487522	1136629	837829	992373	1131621	1278385	1414462	1622432	1212850
	Virtuoso (cold)	46998	77903	74664	82471	109177	86329	79590	74014	167892	78311	81350	159688	95034	109382
	Virtuoso (AM)	10529	16796	13159	17320	17712	18243	15627	9470	19314	10775	10870	18808	15431	14111

	Query	IL-3-5	IL-3-6	IL-3-7	IL-3-8	IL-3-9	IL-3-10	AM-IL-3	AM-5	AM-6	AM-7	AM-8	AM-9	AM-10
SF1000	S2RDF ExtVP	4474	12188	8552	178514	13411	13405	38424	3714	5267	4166	60886	5993	6154
	H2RDF+	121396	183752	225669	F	F	F	N/A	57279	94151	118100	N/A	N/A	N/A
	Sempala	155298	194758	93424	878232	217636	231430	295130	67992	81277	47505	309265	89152	93748
	PigSPARQL	362172	571965	622899	1924061	1653627	1777284	1152001	230868	329447	357796	810268	735277	790757
	SHARD	1323657	2423349	F	F	F	F	N/A	553109	936708	N/A	N/A	N/A	N/A
SF10000	S2RDF ExtVP	29590	87525	102971	2068100	158595	141940	431454	27774	37684	44072	699454	63462	57329
	H2RDF+	240339	451390	F	F	F	F	N/A	131397	221988	N/A	N/A	N/A	N/A
	Sempala	493016	595152	365868	5649620	2026680	2462137	1932079	227782	263319	194362	1959502	759324	904333
	PigSPARQL	1847039	3353907	4876005	40140420	37353210	37514308	20847481	771646	1312782	1836609	13609145	12699271	12764532
	SHARD	11995677	23164293	F	F	F	F	N/A	4541903	8360736	N/A	N/A	N/A	N/A
	Virtuoso (cold)	F	F	F	F	F	F	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Virtuoso (AM)	F	F	F	F	F	F	N/A	N/A	N/A	N/A	N/A	N/A	N/A

predicates in the dataset (`friendOf` and `follows`) that both together represent ca. 70% of all triples in the RDF graph ($0.7 * |G|$). This is probably also the reason why H2RDF+ does not use centralized execution for all queries. By means of ExtVP, S2RDF can reduce the input size for predicates `follows` and `friendOf` from $0.3 * |G|$ to $0.07 * |G|$ and from $0.4 * |G|$ to $0.065 * |G|$ using a combination of OS and SO tables. The only system that achieves similar runtimes as S2RDF is centralized Virtuoso with warm caches while being significantly slower for cold caches.

The second query type (IL-2) has the same structure but starts from an instantiated retailer. Similar to IL-1, S2RDF clearly outperforms all other distributed competitors. However, there is an interesting aspect when comparing the runtime of S2RDF for IL-2-5 with runtimes of IL-2-6 to IL-2-10. IL-2-5 performs much worse than all other queries despite being a subpattern of them. This seems to be odd at first glance but can be explained by looking at the query pattern. IL-2-5 ends with following predicate (edge) `friendOf` twice `(... . ?v3 friendOf ?v4 . ?v4 friendOf ?v5)` which is the largest predicate in the graph ($0.4 * |G|$). Unfortunately, SO table $ExtVP^{SO}_{friendOf|friendOf}$ has selectivity $SF = 1$ and thus table $VP_{friendOf}$ must be used for the last triple pattern t_5. IL-2-6 adds a triple pattern with predicate `likes` to the end which allows S2RDF to use $ExtVP^{OS}_{friendOf|likes}$ with $SF = 0.24$ for t_5 which reduces the input size for it from $0.4 * |G|$ to $0.1 * |G|$. This example demonstrates that more triple patterns in a query can even lead to better performance in S2RDF despite the fact that it needs more joins.

The third query type (IL-3) has also the same structure but is not bound at all, i.e. it starts from all vertices in the graph following various edges. This type of query puts a heavy load on the system and produces very large result sets. Only S2RDF, Sempala and PigSPARQL were able to answer all queries up to diameter 10 on the largest dataset (SF10000) which demonstrates the excellent scalability of these systems. Virtuoso was not able to answer any of the queries within a 10 hours timeout which confirms the scalability limitations of centralized RDF stores. Interesting to see is that S2RDF runtime of IL-3-8 is an order of magnitude slower than for all other queries. There is a twofold reason for that. First, the result size of IL-3-8 is extremely large ($\sim$ 25 billion) and gets reduced by adding another triple pattern t_9 in IL-3-9 ($\sim$ 1 billion). Second, join order optimization (cf. Section 5.2) does not simply add the join for t_9 in IL-3-9 to the end of execution but pushes it more to the beginning as its corresponding ExtVP table is rather small and thus reduces the intermediate result size significantly.

Overall, S2RDF demonstrates its superior performance and scalability for queries with large diameters, a query type that is not sufficiently covered by the Basic Testing use case and is also underrepresented in other existing RDF/SPARQL benchmarks. Our new use case reveals that performance of many distributed RDF stores significantly drops for such workloads as their data model is optimized to answer small diameter queries. It is now included in WatDiv and we hope that this will accelerate a better support in future.

6.3 SF Threshold

In the previous two experiments we did not specify a selectivity threshold for ExtVP (SF TH), i.e. all tables with $SF < 1$ were available to S2RDF for query execution. In this section we investigate the influence of a threshold as discussed in Section 4.3 on query runtime and storage overhead. For example, a threshold of 0.5 for SF means that only those ExtVP tables with $SF < 0.5$ are materialized. Since WatDiv data is synthetic and thus potentially more structured than real-world RDF datasets [8], we additionally used a dump of YAGO [16] (YAGO2s 2.5.3) for this experiment with a total size of 245 million triples. YAGO is a huge semantic knowledge base derived from Wikipedia, WordNet and GeoNames and was also used by IBM Watson.

Table 4: ExtVP sizes (including VP) for varying SF threshold values (WatDiv SF10000 / YAGO)

	SF TH	#tables (%)	#tuples (%)	HDFS size (%)
WatDiv	**0.00**	86 (0.04)	1091 M (0.08)	6.6 GB (0.09)
	0.25	1275 (0.60)	3316 M (0.25)	18.4 GB (0.26)
	0.50	1609 (0.76)	5889 M (0.45)	32.1 GB (0.46)
	0.75	1887 (0.89)	9480 M (0.73)	51.3 GB (0.73)
	1.00	2129 (1.00)	13059 M (1.00)	70.3 GB (1.00)
YAGO	**0.00**	104 (0.02)	245 M (0.09)	2.4 GB (0.10)
	0.25	4977 (0.76)	768 M (0.29)	6.3 GB (0.26)
	0.50	5369 (0.81)	1086 M (0.41)	9.8 GB (0.41)
	0.75	5645 (0.86)	1353 M (0.52)	12.8 GB (0.53)
	1.00	6588 (1.00)	2619 M (1.00)	24.2 GB (1.00)

Table 4 gives an overview of the number of tables maintained by S2RDF (ExtVP + VP), the total number of table tuples and total storage size in HDFS. SF TH = 0 means that no ExtVP tables are stored at all and hence only VP is used whereas SF TH = 1 means that all tables with $SF < 1$ are considered. The best performance, of course, is achieved for SF TH = 1 but this also means that we store a lot of tables with low selectivities that cost a lot of storage but do not improve performance to a large extent.

We test the runtimes of S2RDF for varying SF TH values. For WatDiv we run the Basic Testing use case (cf. Section 6.1) which covers all query shapes. For YAGO we adapted the queries used in [5] (derived from [21]) and [31]. We omit unbound predicate queries as they are not suited to demonstrate the properties of ExtVP. Moreover, we designed additional queries such that the final set (15 queries with up to 13 triple patterns) also covers all shapes similar to WatDiv. Our YAGO query set is available on the project website. A comparison of relative runtimes and actual data sizes with respect to SF TH values is illustrated in Figure 7.

It turns out that for SF TH = 0.25 we already achieve most of the performance benefit compared to the baseline execution with SF TH = 0 (which corresponds to VP), both for synthetic and real-world data. In fact, on average we achieve 95% of best possible performance benefit for WatDiv and even 99% for YAGO. At the same time, we maintain

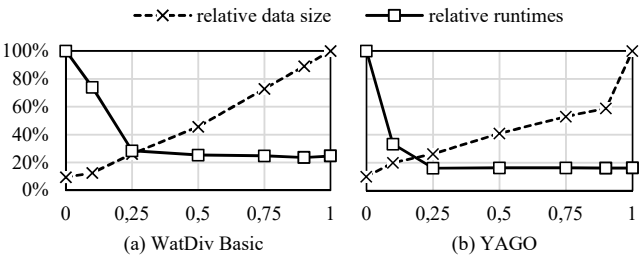

Figure 7: Relative query runtimes and data sizes in relation to increasing SF threshold values

only $3n$ tuples in total over all tables (including VP) with n being the number of triples in the RDF graph, compared to more than $10n$ for SF TH = 1. Thus, ExtVP implies an overhead of only $2n$ compared to VP for SF TH = 0.25 which also directly corresponds to physical storage in HDFS. Most notably, S2RDF performs even better for YAGO than for WatDiv. The reason is that real data is less structured and thus ExtVP tables are more selective on average (YAGO: 0.20 vs. WatDiv: 0.29). Also 76% of all ExtVP tables have $SF < 0.25$ for YAGO compared to 60% for WatDiv. Storage consumption for YAGO leaps up for SF TH > 0.9 because a lot of large-scale tables have $0.9 < SF < 1$ whereas it increases uniformly for WatDiv. This also reveals the more regular nature of WatDiv compared to YAGO. It is also important that there was no significant difference for different query shapes which confirms that our data model does not favor any specific shape. A differentiation by query shape for WatDiv can be found in our technical report [29].

In summary, the performance of S2RDF when executed on ExtVP with a threshold of 0.25 for SF is almost the same compared to ExtVP with no threshold while reducing the overhead significantly to a reasonable extent comparable to existing approaches.

7. CONCLUSION

In this paper, we present S2RDF, a distributed Hadoop-based SPARQL query processor for large-scale RDF data implemented on top of Spark. It comes with a novel relational schema for RDF called ExtVP (Extended Vertical Partitioning) and uses the SQL interface of Spark for query execution by compiling SPARQL to SQL. ExtVP is an extension to the Vertical Partitioning (VP) schema [1] and is inspired by semi-join reductions similar to Join Indices [30]. We precompute the reductions of tables in VP for possible join correlations that can occur between triple patterns in a SPARQL query. In this manner, S2RDF can avoid dangling tuples in the join input which significantly reduces the query input size and thus execution runtime. This technique is applicable to any kind of query pattern regardless of its shape which we demonstrate in our comprehensive evaluation.

S2RDF outperforms state of the art centralized and distributed SPARQL query processors by an order of magnitude on average for all query shapes while achieving subsecond runtimes for the majority of benchmark queries on a billion triples dataset. In contrast to most of the existing distributed RDF stores, the performance of S2RDF using ExtVP does not depend on the query diameter but achieves efficient runtimes on large diameter queries as well. To reduce the overhead compared to VP, one can also specify

an optional threshold for selectivity factor SF of ExtVP. We demonstrate in our evaluation that a threshold of 0.25 achieves 95% of best possible runtime improvements on average while using only 25% of table tuples and storage size.

A more comprehensive discussion, in particular further evaluation results, can be found in our technical report [29].

8. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *PVLDB*, pages 411–422, 2007.

[2] G. Aluc, O. Hartig, M. Özsu, and K. Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *ISWC*, volume 8796 of *LNCS*, pages 197–212, 2014.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[4] P. Bernstein and D.-M. Chiu. Using Semi-Joins to Solve Relational Queries. *J. ACM*, 28(1):25–40, 1981.

[5] R. Binna, W. Gassler, E. Zangerle, D. Pacher, and G. Specht. SpiderStore: A Native Main Memory Approach for Graph Storage. In *GI-Workshop "Grundlagen von Datenbanken"*, volume 733 of *CEUR Workshop Proceedings*, pages 91–96, 2011.

[6] 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*, pages 121–132, 2013.

[7] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *WWW Alt.*, pages 74–83, 2004.

[8] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets. In *SIGMOD*, pages 145–156, 2011.

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

[10] D. C. Faye, O. Cur, and G. Blin. A survey of RDF storage approaches. *Revue Africaine de la Recherche en Informatique et Mathmatiques Appliques*, 15(1):11–35, 2012.

[11] L. Galrraga, K. Hose, and R. Schenkel. Partout: A Distributed Engine for Efficient RDF Processing. In *WWW*, pages 267–268, 2014.

[12] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *SIGMOD*, pages 289–300, 2014.

[13] M. Hammoud, D. A. Rabbou, R. Nouri, S.-M.-R. Beheshti, and S. Sakr. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *PVLDB*, 8(6):654–665, 2015.

[14] S. Harris, N. Lamb, and N. Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. In *SSWS*, volume 517 of *CEUR*, 2009.

[15] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *ISWC*, volume 4825 of *LNCS*, pages 211–224, 2007.

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

[17] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.

[18] Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *The VLDB Journal*, 24(1):67–91, 2015.

[19] A. Kemper and G. Moerkotte. Access support relations: An indexing method for object bases. *Information Systems*, 17(2):117–145, 1992.

[20] H. Kim, P. Ravindra, and K. Anyanwu. From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra. *PVLDB*, 4(12):1426–1429, 2011.

[21] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal*, 19(1):91–113, 2010.

[22] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, 3rd Ed.* Springer New York, 2011.

[23] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF+: High-performance distributed joins over large-scale RDF graphs. In *IEEE BigData*, pages 255–263, 2013.

[24] M.-D. Pham, L. Passing, O. Erling, and P. Boncz. Deriving an Emergent Relational Schema from RDF Data. In *WWW*, pages 864–874, 2015.

[25] J. Prez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.

[26] K. Rohloff and R. E. Schantz. Clause-iteration with MapReduce to Scalably Query Datagraphs in the SHARD Graph-store. In *DIDC*, pages 35–44, 2011.

[27] A. Schätzle, M. Przyjaciel-Zablocki, T. Hornung, and G. Lausen. PigSPARQL: A SPARQL Query Processing Baseline for Big Data. In *ISWC Posters & Demos*, pages 241–244, 2013.

[28] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen. Sempala: Interactive SPARQL Query Processing on Hadoop. In *ISWC*, volume 8796 of *LNCS*, pages 164–179, 2014.

[29] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF Querying with SPARQL on Spark. *CoRR*, arXiv:1512.07021, 2015.

[30] P. Valduriez. Join Indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[31] M. Voigt, A. Mitschick, and J. Schulz. Yet Another Triple Store Benchmark? Practical Experiences with Real-World Data. In *SDA*, volume 912 of *CEUR Workshop Proceedings*, pages 85–94, 2012.

[32] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB*, 1(1):1008–1019, 2008.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28, 2012.