A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data

Ibrahim Abdelaziz[*] Razen Harbi[‡] Zuhair Khayyat[*] Panos Kalnis[*]

[*]King Abdullah University of Science and Technology
{first.last}@kaust.edu.sa

[‡]Saudi Aramco
razen.harbi@aramco.com

ABSTRACT

Distributed SPARQL engines promise to support very large RDF datasets by utilizing shared-nothing computer clusters. Some are based on distributed frameworks such as MapReduce; others implement proprietary distributed processing; and some rely on expensive preprocessing for data partitioning. These systems exhibit a variety of trade-offs that are not well-understood, due to the lack of any comprehensive quantitative and qualitative evaluation. In this paper, we present a survey of 22 state-of-the-art systems that cover the entire spectrum of distributed RDF data processing and categorize them by several characteristics. Then, we select 12 representative systems and perform extensive experimental evaluation with respect to preprocessing cost, query performance, scalability and workload adaptability, using a variety of synthetic and real large datasets with up to 4.3 billion triples. Our results provide valuable insights for practitioners to understand the trade-offs for their usage scenarios. Finally, we publish online our evaluation framework, including all datasets and workloads, for researchers to compare their novel systems against the existing ones.

1. INTRODUCTION

The Resource Description Framework (RDF) [8] is a versatile data model that provides a simple way to express facts in the semantic web. Many large public knowledge bases, including UniProt [10], PubChemRDF [7], Bio2RDF [3] and DBpedia [4] have billions of facts in RDF format. These databases are usually interlinked, and are globally queried using SPARQL [42].

As the volume of RDF data grows, the computational complexity of indexing and querying large datasets becomes challenging. Single-machine RDF systems, like RDF-3X [53] and gStore [37], do not scale well to complex queries on web-scale RDF data [29, 33]. To overcome this problem, many distributed SPARQL query engines [29, 33, 47, 41, 32, 48, 30, 28, 36, 25, 46, 18, 15, 38, 16] have been introduced. They utilize shared-nothing computing clusters and are either built on top of distributed data processing frameworks, such as MapReduce, or implement proprietary distributed computation approaches. They partition the RDF graph among multiple machines (a.k.a., workers) to handle big datasets, and parallelize query execution to reduce the running time. Answering queries typically involve processing of local data at each worker, interleaved with data exchange among workers. The main challenge of distributed systems is scaling-out, which is affected by various factors including data partitioning, load balancing, indexing, query optimization and communication overhead.

Despite the plethora of distributed RDF systems and their practical applications, there is limited information about their comparative performance. Some published surveys provide qualitative comparisons. For example, Sakr et al. [49] present an overview of using the relational model for RDF data. Svoboda et al. [52] classify state-of-the-art indexing approaches for linked data. Kaoudi et al. [34] survey RDF data management systems designed for the cloud. Ozsu [43] provides a broader overview of existing centralized and distributed RDF engines and discusses querying techniques for linked data. Finally, Ma et al. [40] survey techniques that use relational and NoSQL databases to store large RDF data. None of the aforementioned surveys contain any experimental evaluation.

Motivated by the lack of quantitative comparison, in this paper we provide a comprehensive experimental evaluation of state-of-the-art distributed RDF systems, using a variety of very large real datasets and query loads. We start with an extensive survey that covers 22 relevant systems. We describe the execution model and the graph partitioning strategy of each system, discuss the similarities and differences and explain the various trade-offs. We also categorize the systems based on the: (*i*) underlying implementation framework (e.g., MapReduce, vertex-centric or proprietary distributed processing); (*ii*) use of generic joins versus specialized graph exploration; (*iii*) data replication strategy; and (*iv*) adaptivity to query workload.

Then, we perform extensive experimental evaluation of the following 12 representative systems: S2RDF [15], AdPart [46], DREAM [38], Urika-GD [11], CliqueSquare [25], S2X [51], TriAD [48], SHAPE [32], H-RDF-3X [29], H2RDF+ [41], SHARD [47] and gStoreD [45]. We use all the standard synthetic benchmarks (e.g., LUBM [6]) and a variety of very large real datasets (e.g., Bio2RDF [3]) with up to 4.3 billion triples, to stretch the systems to their limits. We analyse the following metrics: (*i*) *Startup cost*: it includes the time overhead of data partitioning, indexing, replication and loading to HDFS/memory, as well as the storage overhead of replication; (*ii*) *Query efficiency*: we utilize query workloads with varying query complexities and selectivities,

Proceedings of the VLDB Endowment, Vol. 10, No. 13
Copyright 2017 VLDB Endowment 2150-8097/17/08.

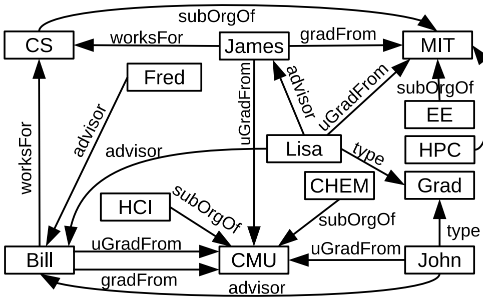


Figure 1: An example RDF graph. Each edge and its associated vertices correspond to an RDF triple; e.g., ⟨`Bill`**, *worksFor*, **`CS`**⟩.**

and report the execution time; (*iii*) *Scalability* to very large datasets; and (*iv*) *Adaptability* to different workloads.

Our results suggest that specialized in-memory systems, such as AdPart [46] or TriAD [48] provide the best performance, assuming the data can fit in the cumulative memory of the computing cluster. If this condition is not satisfied, MapReduce based systems (e.g., H2RDF+ [41]), are an acceptable alternative. In contrast, the startup costs of some systems (e.g., S2RDF [15]) or the excessive replication (e.g., DREAM [38]), severely limit their applicability to large datasets. In an attempt to standardize the evaluation of future systems and assist practitioners to select the appropriate solution for their data and applications, we publish online all datasets, our evaluation methodology and links to the systems.

The rest of the paper is organized as follows: Section 2 provides essential background on RDF and an overview of single machine RDF stores. Section 3 contains a survey of the 22 state-of-the-art distributed RDF systems, whereas Section 4 presents the experimental evaluation of the selected 12 systems. Finally, Section 5 concludes our findings.

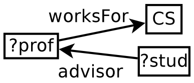
2. BACKGROUND

RDF [8] is a standard data model and the core component of the W3C Semantic Web. RDF datasets consist of triples ⟨`subject`, *predicate*, `object`⟩, where the *predicate* (P) represents a relationship between a `subject` (S) and an `object` (O). RDF data can be viewed as a long relational table with three columns, or as a directed labeled graph with vertices reflecting the entities and edge labels as the predicates. Figure 1 shows an example RDF graph of students and professors in an academic network.

SPARQL [9] is the de-facto query language for RDF data. In its simplest form, a.k.a. Basic Graph Pattern (BGP), a SPARQL query consists of a set of RDF triple patterns; some of the nodes in a pattern are variables that may appear in multiple patterns. For example, the query in Figure 2(a) returns all professors who work for CS with their advisees. The query corresponds to the graph pattern in Figure 2(b). The answer is the set of ordered bindings of $(?prof, ?stud)$ that render the query graph isomorphic to subgraphs in the data. Assuming data are stored in a table $D(s, p, o)$, the query can be answered by first decomposing it into two subqueries: $q_1 \equiv \sigma_{p=worksFor \wedge o=CS}(D)$ and $q_2 \equiv \sigma_{p=advisor}(D)$. The subqueries are answered independently by scanning table D; then, their intermediate results are joined on the subject and object attribute: $q_1 \Join_{q_1.s=q_2.o} q_2$. By applying the query on the data of Figure 1, we get `(?prof, ?stud)` ∈ {`(James, Lisa)`,`(Bill, John)`, `(Bill, Fred)`,`(Bill, Lisa)`}.

```
SELECT ?prof ?stud WHERE {
 ?prof    worksFor  CS .
 ?stud    advisor   ?prof .
}
```

(a) SPARQL



(b) Graph

Figure 2: A SPARQL query that finds CS professors with their advisees.

2.1 Single-machine RDF stores

We discuss the storage models of single-machine RDF systems since several distributed systems [38, 32, 29] rely on them for querying RDF data within a single partition.

Triple Table uses a single table with three columns corresponding to subject, predicate and object to store RDF data. An index is created per column for faster join evaluation. A query with several predicates corresponds to a set of self-joins on the large triple table. This approach scales poorly due to expensive self joins [26]. RDF-3X [53] and Hexastore [23] reduce this cost by using a set of indices that cover all possible permutations of S, P and O. These indices are stored as clustered B+-trees and are compressed using rigorous byte-level techniques. Aggregate indices, selectivity histograms and statistics about frequently accessed paths are used to select the lowest-cost execution plan. Their optimizer use order-preserving merge-joins for most of the operations and hash-joins for the last few ones.

Property table is a wider and flattened representation of RDF data [54]. The dimensions of this table are determined by the number of subjects and distinct predicates. Each cell in the table contains the object value of the corresponding subject and predicate. This representation has high storage overhead when the number of unique predicates is large due to its sparse representation. Moreover, it can not represent multi-valued attributes, i.e., a subject connected to different objects using the same predicate. Jena2 [55] solves this problem by introducing two alternative representations, namely, the clustered property table and property-class tables. BitMat [20] proposes an alternative representation of property tables that uses a compressed 3-dimensions bit-matrix. Each dimension of the bit-matrix corresponds to part of RDF triple; S, P and O. Each cell represents the existence of an RDF triple defined by the S,P,O positions. Queries are executed on the compressed data without materializing intermediate join results.

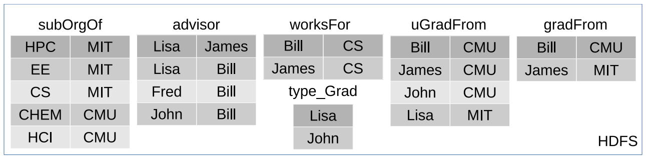
Vertical Partitioning is another representation for RDF data proposed by SW-Store [24]. The triples table is vertically partitioned into n tables, where n is the number of distinct predicates. A two columns table is created for each predicate where a row is a pair of subject-object values connected through the predicate. Tables are sorted on the subject to render subject lookup and merge joins faster. This approach stores multi-valued attributes as successive rows and does not store NULL values. It provides good performance for queries with bounded predicates, however, it requires scanning multiple tables to reconstruct information related to a single entity.

3. DISTRIBUTED RDF SYSTEMS

Distributed RDF systems scale to large datasets by partitioning the RDF graph among many compute nodes (workers) and evaluating queries in a distributed fashion. Each SPARQL query is decomposed into multiple subqueries, which are then evaluated independently. Since the data is

Table 1: Summary of state-of-the-art distributed RDF systems.

System	Partitioning Strategy	Execution Model	MapReduce Based	Graph Based	Specialized System	Replication	Workload Awareness
AdPart [46]	Subject Hash + workload adaptive	Distributed Semi-Join			✓	✓	✓
AdPart-NA [46]	Subject Hash	Distributed Semi-Join			✓		
CliqueSquare [25]	Hybrid (Hash + VP)	MapReduce-based Join	✓				
DREAM [38]	No partitioning; full replication	RDF-3X [53]			✓	✓	
EAGRE [56]	METIS	MapReduce-based Join	✓				
gStoreD [45]	Partitioning Agnostic	gStore [37]		✓	✓		
H-RDF-3X [29]	METIS	RDF-3X [53]	✓		✓	✓	
H2RDF+ [41]	H-Base partitioner (range)	Centralized + MapReduce	✓		✓		
HadoopRDF [30]	VP + predicate files on HDFS	MapReduce Join	✓				
Partout [36]	Workload-based fragmentation	RDF-3X [53]			✓		✓
PigSparql [14]	Hash + Triple-based files	SPARQL to PigLatin	✓				
S2RDF [15]	Extended Vertical Partitioning	SPARQL to SQL	✓			✓	
S2X [51]	GraphX partitioning strategy	Vertex-Centric BGP matching	✓	✓			
Sedge [57]	Subject Hash	Vertex-Centric BGP matching		✓		✓	
Sempala [50]	VP	SPARQL to SQL	✓				
SHAPE [32]	Semantic Hash Partitioning	RDF-3X [53]	✓		✓	✓	
SHARD [47]	Hash	MapReduce-based Join	✓				
TriAD [48]	Hash-based Sharding	Distributed Merge/Hash Joins			✓		
TriAD-SG [48]	METIS + Horizontal Sharding	Distributed Merge/Hash Joins			✓	✓	
Trinity.RDF [33]	Key-value store on graph	Graph Exploration		✓	✓		
WARP [28]	METIS on query workload	RDF-3X [53]	✓		✓	✓	✓



subOrgOf

HPC	MIT
EE	MIT
CS	MIT
CHEM	CMU
HCI	CMU

advisor

Lisa	James
Lisa	Bill
Fred	Bill
John	Bill

worksFor

Bill	CS
James	CS

type_Grad

Lisa
John

uGradFrom

Bill	CMU
James	CMU
John	CMU
Lisa	MIT

gradFrom

Bill	CMU
James	MIT

Figure 3: Vertical partitions of the data of Fig. 1.

distributed, nodes may need to exchange intermediate results during query evaluation; therefore queries with large intermediate results incur high communication cost [48, 29]. To achieve acceptable response time, distributed systems attempt to minimize communication cost and maximize parallelism. This is accomplished by efficient data partitioning that maximizes data locality; load balancing that avoids stragglers; efficient join implementations; and the utilization of native RDF indexing techniques to speedup joins and index lookups. We summarize in Table 1 the various features of existing distributed RDF systems. Note that `Replication` in this table points to systems that explicitly replicate RDF data, excluding HDFS replication.

In this survey, we categorize distributed RDF management systems along 2 dimensions based on their execution model: (i) MapReduce and Graph-based systems: such systems rely on general purpose frameworks, i.e., Hadoop or Spark, that offer seamless data distribution and parallelization at the cost of flexibility. (ii) Specialized RDF systems: are built specifically for SPARQL query evaluation by utilizing custom physical layouts, native RDF indexing, efficient communication protocols and explicit replication. Within the second category, we define three subcategories based on the data partitioning scheme: lightweight, sophisticated and workload-aware partitioning. Systems based on sophisticated partitioning offer faster query execution at the cost of startup time and storage requirements. Workload-aware systems achieve faster query execution by adapting their data partitioning to the entire query workload.

3.1 MapReduce and Graph Based Systems

SHARD [47] is a triple store implemented on top of MapReduce [31]. The entire RDF dataset is stored in a single file within HDFS [2], where each line represents all triples of a single subject. The input dataset is hash-partitioned among workers such that each worker is responsible for a distinct set of triples. SHARD does not use indexing; consequently, during query evaluation it scans the entire dataset. SPARQL queries are executed as a sequence of MapReduce iterations. Each iteration is responsible for a single subquery, while the results are continuously joined with subsequent iterations. The final iteration is responsible for filtering the bounded variables and removing redundant results.

HadoopRDF [30] also uses HDFS to store the RDF data as flat files; replication and distribution is left to HDFS. Unlike SHARD, HadoopRDF uses multiple files on HDFS, a file for each predicate which is similar to SW-Store's [24] vertical partitioning. HadoopRDF also splits each predicate file into multiple smaller files based on explicit and implicit type information. Initially, it divides the *rdf:type* file into as many files as the number of distinct objects. Then, a set of files are created for each type *type_object*. For example, *rdf:type* in Figure 3 results in only one file (*type_Grad*) because predicate *type* has only one object (*Grad*). Then, HadoopRDF divides the remaining predicate files into multiple files based on the object type. It splits triples based on the RDF class their object belongs to. For example, triple ⟨`John`, $teacherOf$, `OS`⟩ will be stored in a file named *teacherOf_Course*. To retrieve this implicit type information, it needs to join the predicate file with the type_* files. HadoopRDF executes queries as a sequence of MapReduce iterations. It contains a query optimizer to select the query plan that minimizes the number of MapReduce iterations and the size of intermediate results.

CliqueSquare [25] exploits the replication of HDFS to maximize the efficiency of parallel joins and to reduce the communication cost. To achieve this, CliqueSquare partitions and stores the data in three different ways, by hashing each triple on the subject, predicate and object. Furthermore, it partitions the data within each machine into smaller files by applying property-based grouping similar to HadoopRDF. The integration of HDFS replication and partitioning enables CliqueSquare to perform all first-level joins (i.e., subject-subject, subject-predicate, etc.) locally on each machine. The CliqueSquare optimizer minimizes the number of joins by generating relatively shallow plans that use multi-way joins. It finds the possible clique decompositions of the query graph, where each clique corresponds to a multi-way join, and then selects the decomposition with the lowest cost. The resulting plan is executed as a sequence of MapReduce iterations.

H2RDF+ [41] is a distributed RDF engine based on MapReduce and Apache HBase [5]. It materializes all six permutations of RDF triples using HBase tables, which are sorted key-value maps. Data partitioning is left to HBase that range-partitions tables based on keys. Maintaining these indices offers several benefits: (*i*) all SPARQL triple patterns can be answered efficiently by a single scan on the corresponding index; (*ii*) merge join can be employed to exploit the precomputed ordering in these indices; and (*iii*) every join between triple patterns is executed as merge-join. H2RDF+ maintains a set of aggregated statistics to estimate the selectivity of triple patterns, join results and join cost. It uses a greedy algorithm that finds at each execution step the join with the lowest cost. It uses multi-way merge join for sorted data, and sort-merge join for unsorted intermediate results. Simple queries are executed efficiently in a centralized fashion, while complex queries with large intermediate results are evaluated as a sequence of MapReduce jobs. H2RDF+ utilizes lazy materialization to minimize the size of the intermediate results.

S2X [51] exploits the inherited graph structure of RDF to process SPARQL as graph-based computations on top of GraphX. It uses the parallel vertex-centric model to evaluate the BGP matching of SPARQL while other operators, such as `OPTIONAL` and `FILTER`, are processed through Spark RDD operators. BGP matching starts by distributing all triple patterns to all graph vertices. Each vertex matches its edge labels with the triple's predicate. Graph vertices cooperatively validate their triple candidacy with their direct neighbours by exchanging messages. Then, the partial results are collected and incrementally merged. S2X uses two string encoding types: hash and count-based. Hash-based encoding utilizes a 64-bit hash function to encode subjects and objects, while count-based assigns unique numeric values to them. S2X does not have a special RDF partitioner; it uses GraphX 2D hashing, which hashes on the encoded subject then hashes on the encoded object, to partition the input graph among workers.

Sedge [57] proposes similar techniques for SPARQL query execution on top of the vertex-centric processing model. The entire graph is replicated several times and each replica is partitioned differently. Each SPARQL query is executed against the replica that minimizes communication. Sedge does not provide automatic translation of SPARQL queries to the vertex-centric model; a vertex-centric program has to be written manually for each query, which is counterproductive and requires prior knowledge about the data.

3.2 Specialized RDF Systems

3.2.1 *Lightweight Partitioning*

Trinity.RDF [33] is a distributed in-memory RDF engine that can handle large datasets. It represents RDF data in a graph form using adjacency lists, stored in the Trinity key-value store [21]. The graph is hash-partitioned on vertex-id; this is equivalent to partitioning the data twice, on subject and object. Trinity.RDF uses graph exploration for query evaluation. In every iteration, a single subquery is explored starting from the valid bindings in all workers. For example, Figure 4 shows how Trinity.RDF executes Q_{prof}. Starting with the pattern ⟨`?prof`, *worksFor*, `CS`⟩, it explores the neighbours of CS connected via $worksFor$. It finds that the possible binding for *?prof* are James and Bill. In the next iteration, it starts to explore from nodes James and Bill via edge *advisor*, and generates the bindings for ⟨`?stud`, *advisor*, `?prof`⟩. Graph exploration avoids the generation of redundant intermediate results. However, because exploration only involves two vertices (source and target), Trinity.RDF cannot prune invalid intermediate results without carrying all their historical bindings. Hence, workers need to ship candidate results to the master to finalize processing, which is a potential bottleneck.

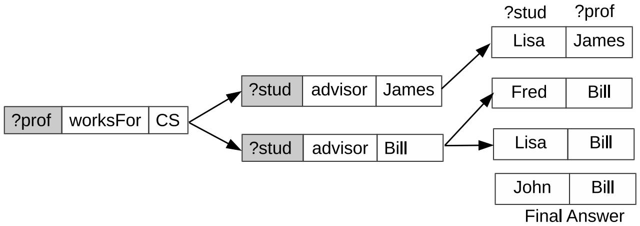


Figure 4: Trinity.RDF Graph Exploration Execution Plan for Q_{prof} using the data in Figure 2.

TriAD [48] employs lightweight hash partitioning based on both subjects and objects. Since partitioning information is encoded into the triples, TriAD has full locality awareness of the data and processes large number of concurrent joins without communication. It creates six in-memory tables on each machine, one for each permutation of subject, predicate, object. The six SPO permutations are arranged into two groups; subject-key indices (SPO, SOP, PSO), and object-key indices (OSP, OPS, POS). Each of these indices is then hash-partitioned among different machines and sorted within each machine in lexicographic order. This enables TriAD to perform efficient distributed merge-joins over the different SPO indices. Multiple join operators are executed concurrently by all workers, which communicate via asynchronous message passing. At each compute node, TriAD uses multiple-threads to evaluate multiple operators in the query plan in parallel. TriAD shards one (both) relation(s) when evaluating distributed merge (hash) joins, which does not preserve the locality of intermediate results. This causes TriAD to re-shard intermediate results if the sharding column of the previous join is not the current join column. This cost is significant for large intermediate results with multiple attributes.

AdPart-NA [46] employs lightweight partitioning that hashes triples on subject. Each worker stores its local set of triples using three in-memory data structures; P-index, PS-index and PO-index. P-index returns the set triples having the given predicate. Similarly, PS and PO indices return the nodes connected to the given predicate-subject or predicate-object, respectively. AdPart-NA exploits the query structure and the hash-based data locality in order to minimize the communication cost during query evaluation. It capitalizes on the subject-based locality and the locality of its intermediate results (pinning) to evaluate joins in parallel without communication. Star queries joining on subjects are processed in parallel by all workers. Whenever possible, intermediate results are hash-distributed among workers instead of broadcasting to all workers. Figure 5 shows how AdPart-NA evaluates Q_{prof} by a single subject-object join, assuming the following execution order: q_1: ⟨`?prof`, *worksFor*, `CS`⟩, q_2: ⟨`?stud`, *advisor*, `?prof`⟩. For such

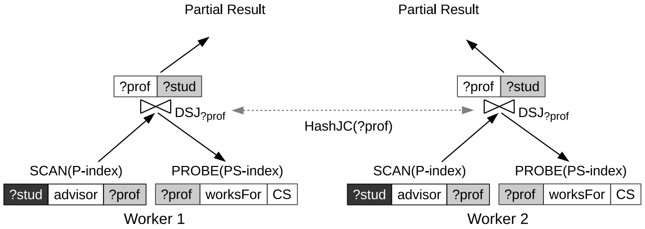


Figure 5: Adpart-NA Query Execution Plan for the SPARQL query in Figure 2.

queries, AdPart-NA employs distributed semi-join. Each worker scans its PO index to find all triples matching q_1, projects on join column $?prof$, and exchanges it with the other worker. Once the projected column is received, each worker computes semi-join $q_1 \rtimes_{?prof} q_2$ using its PO index. Specifically, w_1 probes $p = \texttt{advisor}, o = \texttt{Bill}$ while w_2 probes $p = \texttt{advisor}, o = \texttt{James}$. Finally, each worker computes $q_1 \bowtie_{?prof} q_2$.

DREAM [38] utilizes data replication instead of partitioning by building a single database that is replicated to all workers. It also avoids expensive intermediate data shuffling and only exchanges small auxiliary data. Each machine uses RDF-3X on its assigned data for statistics estimation and query evaluation. DREAM decomposes each query into multiple, usually non-overlapping, subqueries where each subquery is answered by a single worker. Depending on the query complexity, DREAM's optimizer decides to run it either in a centralized or in a distributed fashion. Although DREAM does not incur any partitioning overhead, it exhibits excessive replication and costly preprocessing because of the centralized database construction.

gStoreD [45] is a distributed partitioning-agnostic system. It does not decompose the input query which is sent as-is to all workers. gStoreD starts the query evaluation by computing the partial local matches at each worker. This process depends on a revised version of gStore [37], a single-machine graph-based RDF engine. Then, gStoreD assembles the partial matches to build the cross-partition results. gStoreD allows for two modes of assembly; centralized and distributed. The centralized assembly mode sends the partial results to a centralized site, whereas distributed mode assembles them in multiples sites in parallel.

3.2.2 *Sophisticated Partitioning*

H-RDF-3X [29] uses METIS [27], a balanced vertex partitioning method, to efficiently assign each graph node to a single partition. Then, H-RDF-3X enforces the so-called k-hop guarantee where, for any vertex v assigned to partition p, all vertices up to k-hops away and the corresponding edges are replicated in p. This way any query within radius k can be executed without communication. For example, partitioning the graph in Figure 1 among two workers using 1-hop undirected guarantee yields the partitions shown in Table 2. Each partition is stored and managed by a standalone centralized RDF-3X store; duplicate results are expected due to replication. For example, query $Q =$⟨`?stud`, *advisor*, `Bill`⟩ returns duplicate ⟨`Lisa`, *advisor*, `Bill`⟩ and ⟨`Fred`, *advisor*, `Bill`⟩; one from each partition. To solve this problem, H-RDF-3X introduces the notion of triple ownership. For each vertex v assigned to partition p, H-RDF-3X stores a new triple ⟨`v`, *is_owned*, `yes`⟩ at partition p. During query evaluation an extra join is required for filtering out duplicate results. Queries with radius larger than k are executed using expensive MapReduce joins. k must be small (e.g., $k \leq 2$ in [29]) because replication increases exponentially with k.

Table 2: 1-hop undirected guarantee partitioning of the RDF in Figure 1. `subject(s)` and `object(s)` highlighted in blue are owned by W1; the rest belong to W2. Replicated triples are highlighted in yellow.

W1			W2		
subject	predicate	object	subject	predicate	object
HPC	subOrgOf	MIT	CS	subOrgOf	MIT
EE	subOrgOf	MIT	HCI	subOrgOf	CMU
CHEM	subOrgOf	CMU	Bill	worksFor	CS
James	worksFor	CS	Bill	gradFrom	CMU
James	uGradFrom	CMU	Bill	uGradFrom	CMU
James	gradFrom	MIT	John	type	Grad
Lisa	uGradFrom	MIT	John	uGradFrom	CMU
Lisa	type	Grad	John	advisor	Bill
Lisa	advisor	James	CHEM	subOrgOf	CMU
Lisa	advisor	Bill	James	uGradFrom	CMU
Fred	advisor	Bill	Lisa	advisor	Bill
John	type	Grad	James	worksFor	CS
CS	subOrgOf	MIT	Fred	advisor	Bill

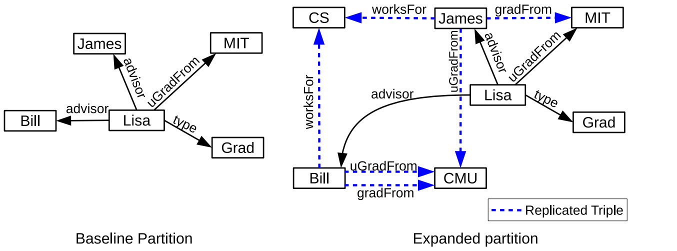


Figure 6: An example of semantic hash partitioning with forward direction and $k = 1$.

EAGRE [56] transforms the RDF data into an entity graph by grouping triples based on the subject where each subject is called an entity. Then, it groups entities with similar properties into an entity class. EAGRE generates a compressed entity graph containing only the entity classes and their relationships which is then partitioned using METIS. At each machine, entities belonging to the same class are treated as high dimensional data indexed by a Space Filling Curve. This maintains an order preserving layout of the data which fits well range and order by queries. EAGRE aims at minimizing the I/O costs by a distributed scheduling approach that reduces the total number of data blocks to read for query evaluation. Similar to H-RDF-3X, EAGRE also suffers from the overhead of MapReduce joins for queries that cannot be evaluated locally.

SHAPE [32] uses semantic hash partitioning to group vertices based on URI hierarchy for the sake of increasing data locality. It identifies groups of triples anchored at the same subject or object and tries to place these grouped triples in the same partition. Then, SHAPE applies its semantic hashing technique in two phases: (i) baseline hash partitioning and (ii) $k - hop$ expansion which adds to each partition all triples whose shortest distance to any anchor of the partition is at most k. Figure 6 shows an example for how to expand a baseline partition in the forward direction with $k = 1$. Each resulting partition is managed by a standalone RDF-3X store. Similar to H-RDF-3X, SHAPE suffers from the high overhead of MapReduce joins. It also requires an extra join for filtering duplicate results. Furthermore, URI-based grouping results in skewed partitioning if a large percentage of vertices share prefixes.

TriAD-SG [48] is a variation of TriAD that uses METIS for data partitioning. Edges that cross partitions are replicated,

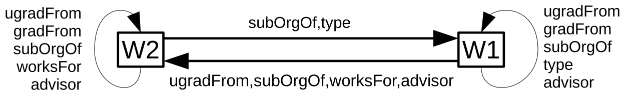


Figure 7: A summary graph for the RDF in Figure 1.

resulting in 1−*hop* guarantee. It defines a summary graph which includes a vertex for each partition; edges connect vertices that share cross-partition edges. Figure 7 shows the summary graph of the data in Figure 1. Queries in TriAD-SG are evaluated against the summary graph first, in order to prune partitions that do not contribute to query results. Then, they are evaluated on the RDF data residing in the partitions retrieved from the summary graph. Multiple join operators are executed concurrently by all workers, which communicate via an asynchronous message passing protocol.

S2RDF [15] is a SPARQL engine built on top of Spark [39]. It proposes a relational partitioning technique for RDF data called Extended Vertical partitioning (ExtVP). ExtVP extends the vertical partitioning approach used by HadoopRDF to minimize the size of input data during query evaluation. ExtVP uses semi-join reduction [44] to minimize data skewness and eliminate dangling triples that do not contribute to any join. For every two vertical partitions (see Figure 3), ExtVP pre-computes join reductions. The results are materialized as tables in HDFS. Specifically, for two partitions P_1 and P_2, S2RDF pre-computes: (*i*) subject-subject: $P_1 \ltimes_{s=s} P_2$, $P_2 \ltimes_{s=s} P_1$, (*ii*) subject-object: $P_1 \ltimes_{s=o} P_2$, $P_2 \ltimes_{s=o} P_1$, and (*iii*) object-subject: $P_1 \ltimes_{o=s} P_2$, $P_2 \ltimes_{o=s} P_1$. The objective of this reduction is to use the semi-join reduced tables for joins instead of the base table since the reduced tables are much smaller, i.e., $T_1 \bowtie_{A=B} T_2 = (T_1 \ltimes_{A=B} T_2) \bowtie (T_1 \rtimes_{A=B} T_2)$. S2RDF does not run on Spark directly; it translates SPARQL queries into SQL jobs which are then executed on top of Spark SQL [19]. S2RDF follows a similar approach to **Sempala** [50] and **PigSPARQL** [14]. Sempala is a distributed RDF engine that translates SPARQL into SQL which runs on top of Apache Impala [35]. Similarly, PigSPARQL translates SPARQL queries into Pig Latin [22] scripts on Apache Pig.

3.2.3 Workload-Aware Partitioning

Partout [36] is a workload-aware distributed RDF engine. It relies on a given query workload to extract representative triple patterns and uses them to partition the data into fragments. Partout has two objectives: (*i*) collocate fragments that are used together in queries; and (*ii*) achieve load balancing among workers. Partout defines a load score for each fragment and sorts fragments in descending order. For each fragment, it calculates a benefit score for allocating it to each machine. The benefit score takes into account both the machine utilization well as the fragment locality. Each worker runs RDF-3X on its assigned fragments. Partout uses global statistics to generate an initial query plan, which is then refined by a cost model that considers data fragments and their locations. The final query plan is executed in parallel by all machines, where each machine sends the results to other hosts in the pipeline.

WARP [28] uses a representative query workload to replicate frequently accessed data by extending the $n-hop$ guarantee method [29]. Given a user query, WARP determines its center node and radius. If the query is within the $n-hop$ guarantee, WARP sends the query to all machines, which evaluate the query in parallel. Otherwise, the query is decomposed into subqueries for which a distributed query evaluation plan is created. Subqueries are evaluated in parallel by all machines and the results are sent to the master which combines them using merge join.

AdPart [46] extends AdPart-NA with adaptive workload-awareness, to cope with the dynamism of RDF workloads. It monitors the query workload and incrementally redistributes parts of the data that are frequently accessed by hot patterns. By maintaining these patterns, many future queries are evaluated without communication. The adaptivity of AdPart complements its good performance on queries that can benefit from its hash-based data locality. Frequent query patterns that are not favored by the initial partitioning (e.g., star joins on an object) are processed in parallel.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of 12 representative distributed RDF systems using multiple real and synthetic datasets. All datasets, workloads, and the detailed results of our experiments are available online [1].

4.1 System Setup

Datasets: We use real and synthetic datasets of variable sizes, summarized in Table 3. We utilize the LUBM [6] synthetic data generator to create a dataset of 10,240 universities consisting of 1.36 billion triples. LUBM and its template queries are widely used for testing most distributed RDF engines [33, 41, 32, 48]. We also use WatDiv [12] which is a recent benchmark that provides a wide spectrum of queries with varying structural characteristics and selectivity classes. We consider two versions of this synthetic dataset: WatDiv with 109 million and WatDiv-1B with 1 billion triples. We also use two real datasets; YAGO2 [13] and Bio2RDF [3]. YAGO2 is derived from Wikipedia, WordNet and GeoNames containing 284 million triples. Bio2RDF provides linked data for life sciences and contains around 4.3 billion triples connecting 24 different biological datasets.

Hardware Setup: All systems are deployed on a 12 machine cluster connected by a 10Gbps Ethernet switch. Each machine has a 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs (12 cores each). The cluster runs a 64-bit Linux.

Hadoop and Spark configuration: We use Hadoop 1.2.1 and Spark 1.6.2 for MapReduce and Spark-based systems and configure them to best utilize the available resources. For Hadoop, we use 12 mappers per worker (a total of 144 mappers), while we configure Spark with 12 cores per worker to achieve a total of 144 Spark cores. Note that we do not use all 24 cores for Hadoop and Spark workers to allow for other background processes, such as HDFS threads and Spark communicator threads.

Compared Systems: We evaluate the performance of 5 systems from the MapReduce and Graph-based category and 7 from Specialized RDF systems. The 12 evaluated systems are: (*i*) AdPart [46], the current state-of-the-art distributed RDF engine. (*ii*) TriAD [48], a recent efficient in-memory RDF system that uses lightweight hash partitioning. We also consider TriAD-SG, which uses graph summaries for join-ahead pruning. (*iii*) Three Hadoop-based systems which use lightweight partitioning: CliqueSquare [25], H2RDF+ [41] and SHARD [47]. (*iv*) SHAPE [32], a semantic hash partitioning approach for RDF data. (*v*) H-RDF-3X [29], a system that uses METIS for graph partitioning and applies the $k - hop$ guarantee scheme. We configure SHAPE with full level semantic hash

Table 3: Datasets; M: millions. #S, #P, #O denote number of distinct subjects, predicates and objects.

Dataset	Triples (M)	#S (M)	#O (M)	#S∩O (M)	#P	Size (GB)	Indegree (Avg/StDev)	Outdegree (Avg/StDev)
WatDiv	109.23	5.21	17.93	4.72	86	15	22.49/960.44	42.20/89.25
YAGO2	284.30	10.12	52.34	1.77	98	42	5.43/2962.93	28.09/35.89
WatDiv-1B	1,092.16	52.12	179.09	46.95	86	149	23.69/2783.40	41.91/89.05
LUBM-10240	1,366.71	222.21	165.29	51.00	18	224	16.54/26000.00	12.30/5.97
Bio2RDF	4,287.59	552.08	1,075.58	491.73	1,714	596	8.64/21110.00	16.83/195.44

Table 4: Partitioning config. and replication ratio.

	(a) Partitioning Config.		(b) Initial Replication		
	H-RDF-3X	SHAPE	H-RDF-3X	SHAPE	DREAM
LUBM-10240	2 undirected	2 forward	19.5%	42.9%	1200%
WatDiv	3 undirected	3 undirected	1090%	0%	1200%
YAGO2	2 undirected	2 forward	73.7%	0%	1200%
Bio2RDF	2 undirected	2 undirected	N/A	N/A	1200%

Table 5: Preprocessing time (Minutes).

	LUBM-10240	WatDiv	YAGO2	Bio2RDF
	Single Machine Systems			
gStore	>24h	108	166	>24h
RDF-3X	392	33	91	>24h
	Distributed Systems			
SHARD	72	9	17	143
H2RDF+	152	9	22	387
CliqueSquare	167	10	19	N/A
S2X (Count)	48	3	9	158
S2X (Hash)	114	8	N/A	N/A
S2RDF (VP)	84	13	25	>24h
S2RDF (ExtVP)	204	225	1,144	>24h
AdPart-NA	14	1.2	4	29
TriAD	72	4	11	75
TriAD-SG	737	63	N/A	N/A
H-RDF-3X	939	285	199	>24h
SHAPE	263	79	251	>24h
gStoreD	>24h	85	254	>24h
DREAM	427	36	98	>24h

partitioning and enable the type optimization. For H-RDF-3X, we enable the type and high degree vertices optimizations. (*vi*) S2RDF [15], a SQL-based RDF engine on top of Spark. (*vii*) S2X [51], an RDF engine on top of GraphX. (*viii*) DREAM [38] which distributes the query execution among fully-fledged unpartitioned data stores. (*ix*) Urika-GD; a data analytics appliance which provides an RDF triplestore and a SPARQL query engine. Urika-GD provides graph-optimized hardware with 2TB of global shared-memory and 64 Threadstorm processors with 128 hardware threads per processor. (*x*) gStoreD [45] a partitioning agnostic approach for distributed SPARQL query processing. Finally, as baselines, we also compare to two single-machine engines; (*xi*) RDF-3X [53] and (*xii*) gStore [37]. These 12 systems were selected based on: (*a*) the availability of their source codes, (*b*) they were successfully run on our cluster and provided the correct results, and (*c*) they either achieve the best performance in their category or introduce novel approaches for distributed SPARQL query evaluation.

Configuration: We use the source codes provided by each system's authors and enable all optimizations. H-RDF-3X and SHAPE were configured to partition each dataset such that all queries are processable without communication (Table 4(a)). To achieve this, we configure H-RDF-3X with undirected guarantee, instead of forward guarantee. For TriAD-SG, we use the same number of partitions reported in [48] for LUBM-10240 and WatDiv. Determining the number of summary graph partitions requires empirical evaluation of some data workload or a representative sample. Generating a representative sample from real data is tricky, whereas empirical evaluation on the original data is costly. Therefore, we do not evaluate TriAD-SG on Bio2RDF and YAGO2. For experiments on individual queries, the query performance is averaged over five runs for each system. The query workloads in scalability and workload adaptivity experiments are evaluated only once for each system.

4.2 Startup Overhead

Our first experiment measures the time it takes all systems for preparing the data prior to answering queries. In Table 5, systems are only allowed 24 hours to complete preprocessing. For a fair comparison, we include the overhead of loading data into HDFS for Hadoop and Spark-based systems; however, we exclude the string-to-id mapping time for all systems.

Lightweight partitioning: The preprocessing phase of systems under this category require the least time due to their lightweight partitioning overhead. SHARD and H2RDF+ employ random and range-based partitioning, respectively, while CliqueSquare uses a combination of hash and vertical partitioning. S2X depends on GraphX's default partitioning strategy, however, its encoding techniques may affect GraphX's partitioning results. The hash-based encoding has very long loading time and cannot load all graphs, such as YAGO2. On the other hand, count-based encoding has faster data loading in GraphX but is slightly slower in query runtime. In our experiments, we only report the best query evaluation results for S2X irrespective of the encoding scheme since their performance do not vary significantly. MapReduce-based systems suffer from the overhead of storing their data on HDFS first before they can start their preprocessing phase. Both TriAD and AdPart-NA use hash-based partitioning. However, TriAD is slower than AdPart-NA because it sorts indices, gathers statistics and partitions its data twice (on subject and object columns).

Sophisticated partitioning: The preprocessing overhead of LUBM-10240 using RDF-3X is significant compared to several distributed engines, whereas gStore failed to preprocess this dataset within 24 hours. Both systems could preprocess WatDiv-100 and YAGO2 in a reasonable time as these datasets are much smaller than LUBM-10240. However, they are still slower than several distributed systems such as AdPart-NA and H2RDF+. Both single-machine based systems failed to preprocess Bio2RDF within 24 hours. As Table 5 shows, systems that rely on METIS for partitioning (i.e., H-RDF-3X and TriAD-SG) have significant startup cost since METIS does not scale to large RDF graphs [32]. To apply METIS, we had to remove all triples connected to literals; otherwise, METIS will take several days to partition LUBM-10240 and YAGO2. For LUBM-10240, SHAPE incurs less preprocessing time compared to METIS-based systems. However, SHAPE performs worse for WatDiv and YAGO2 due to data imbalance, causing some of its RDF-3X engines to take more time in data indexing. Due to the uniform URI's of YAGO2 and WatDiv, SHAPE could not utilize its semantic hash partitioning and placed the entire dataset in a single partition. Finally, both SHAPE and H-RDF-3X did not finish partitioning Bio2RDF and were terminated after 24 hours.

S2RDF has two preprocessing modes: VP and ExtVP. VP mode partitions the RDF graph based on the triples predicate, and stores each predicate in HDFS as a compressed columnar storage format (parquet file). ExtVP mode builds on top of VP. For every two VPs, ExtVP pre-computes its join reductions and materializes the results as new partitions (tables for Spark SQL) in HDFS. Hence, ExtVP in-

Table 6: Runtime for LUBM-10240 queries (ms). SM: Single Machine, MR: MapReduce, and SS: Specialized systems. S2X failed to execute all queries; gStore and gStoreD could not preprocess the data within 24 hours.

	LUBM-10240	Complex Queries				Simple Queries			Geo-	Query/h
		L1	L2	L3	L7	L4	L5	L6	Mean	
SM	RDF-3X	1,812,250	101,750	1,898,500	98,250	38	20	526	10,466	6
MR	SHARD	413,720	187,310	N/A	469,340	358,200	116,620	209,800	261,362	N/A
	H2RDF+	285,430	71,720	264,780	180,320	24,120	4,760	22,910	59,275	30
	CliqueSquare	125,020	71,010	80,010	224,040	90,010	24,000	37,010	74,488	39
	S2RDF-VP	217,537	28,917	145,761	29,965	46,770	5,233	11,308	35,845	52
	S2RDF-ExtVP	46,552	35,802	21,533	47,343	9,222	2,718	4,555	15,275	150
SS	AdPart-NA	2,743	120	320	3,203	1	1	40	75	3,920
	TriAD	6,023	1,519	2,387	17,586	6	4	114	369	912
	TriAD-SG	5,392	1,774	4,636	21,567	9	5	10	333	755
	Urika-GD	5,835	2,396	1,871	6,951	1,442	720	1,588	2,259	1,211
	H-RDF-3X	7,004	2,640	7,957	7,175	1,635	1,586	1,965	3,412	841
	H-RDF-3X (in-memory)	6,841	2,597	7,948	7,551	1,596	1,594	1,926	3,397	839
	SHAPE	25,319	4,387	25,360	15,026	1,603	1,574	1,567	5,575	337
	DREAM	13,031,410	98,263	2,358	4,700,381	18	14	10,755	12,110	1
	DREAM (cached)	1,843,376	98,263	<1	83,053	18	14	468	911	12

curs significantly higher overhead, an order of magnitude slower, compared to its VP version. The overhead incurred by ExtVP depends on several factors including dataset size, density and number of predicates. For example, the size of YAGO2 is almost five times less than the size of LUBM-10240, however, ExtVP requires almost an order of magnitude extra time to process YAGO. This behaviour is also noticed with ExtVP on WatDiv in comparison to LUBM-10240. This is mainly due to the sparsity and the lower number of predicates in LUBM-10240 compared to YAGO and WatDiv. In particular, ExtVP generates a total of 179 partitions (30,337 HDFS objects) for LUBM-10240 compared to 2,221 partitions (319,662 HDFS objects) for WatDiv and 8,125 partitions ($\approx$ 2 million HDFS objects) for YAGO2. Due to this high overhead, S2RDF failed to preprocess Bio2RDF dataset within 24 hours.

4.3 Initial Replication Cost

We only report the initial replication for H-RDF-3X, SHAPE and DREAM (Table 4(b)) as other systems do not explicitly apply replication. H-RDF-3X results in a 19.5% replication for LUBM-10240 and a 73.7% for YAGO2. The sparsity and uniformity of LUBM allow METIS to generate a small edge cut between the different partitions of LUBM which significantly reduces the cross partition replication of H-RDF-3X. METIS, however, fails to reduce the graph edge-cuts for WatDiv because of its dense nature. It caused H-RDF-3X to replicate the whole partitions blindly using k-hop guarantee resulting in a 1090% replication. SHAPE incurs 42.9% replication for LUBM due to its full level semantic hash partitioning and type optimization. It also places WatDiv and YAGO2 in a single partition since the URI's of both datasets are uniform. This results in a 0% replication and causes SHAPE to be as slow as a single-machine RDF-3X store for these two datasets. Similarly, DREAM indexes the entire database on a single machine which is then replicated to other workers. DREAM's preprocessing overhead is reasonable for small datasets (Table 5), however; it does not scale for larger graphs where it requires more than a day to build the Bio2RDF database. For our 12 machines cluster, the replication ratio of DREAM is 1200% for all datasets.

4.4 Query Performance

4.4.1 LUBM dataset

In this experiment (see Table 6), we use the LUBM-10240 dataset and L1-L7 [20] queries. We classify those queries, based on their structure and selectivity into simple and complex. L4 and L5 are simple selective star queries. L6 is also a simple query because it is highly selective. L1, L3 and L7 are complex queries with large intermediate results but very few final results. Finally, L2 is a simple yet non-selective star query that generates large final results.

RDF-3X (single-machine) performs well for simple queries while it is significantly slower for complex queries. SHARD, H2RDF+ and CliqueSquare suffer from the expensive overhead of MapReduce joins; hence, their performance is significantly worse than other systems. For selective simple queries, H2RDF+ avoids the overhead of MapReduce-based joins by solving these queries in a centralized manner which is an order of magnitude faster. The flat plans of CliqueSquare significantly reduce the join overhead for complex queries and achieve up to an order of magnitude better performance. S2X fails to run all queries as it generates a lot of intermediate results at the vertex level. Compared to MapReduce systems, S2RDF-ExtVP shows significant performance improvement due to its in-memory caching technique as well as the materialized join reduction tables. Note that S2RDF requires loading multiple partitions into memory for each query before execution. For example, it loads 6 partitions (1,200 HDFS object) to process L1 and L3.

SHAPE and H-RDF-3X perform better than MapReduce-based systems because they do not require communication. H-RDF-3X performs better than SHAPE as it has less replication. However, as both SHAPE and H-RDF-3X use MapReduce for dispatching queries to workers, they still suffer from the non-negligible overhead of MapReduce (1.5 sec on our cluster). Without this overhead, both systems would perform well for simple selective queries. For complex queries, these systems still perform reasonably well as they run in parallel without any communication overhead. For example, for query L7 which requires excessive communication, H-RDF-3X and SHAPE perform better than some of the specialized systems, such as TriAD. With low hop guarantee, the preprocessing cost for SHAPE and H-RDF-3X can be reduced at the cost of worse query performance because of the MapReduce joins. Although SHAPE and H-RDF-3X do not incur any communication, they still suffer from two limitations: (i) managing the original and replicated data in the same set of indices results in large and duplicate intermediate results, rendering the cost of join evaluation higher and (ii) to filter out duplicate results, H-RDF-3X requires an additional join with the ownership triple pattern. For fairness, we also stored H-RDF-3X databases in a memory-mounted partition; still, it did not affect the performance significantly. As a result, we do not consider in-memory H-RDF-3X in further experiments.

Table 7: Query runtimes (ms) for WatDiv and YAGO2 datasets. SHARD and gStoreD crashed in several queries while DREAM could not finish WatDiv queries within 24 hours.

		WatDiv-100 (GeoMean)				**YAGO2**					
		L1-L5	**S1-S7**	**F1-F5**	**C1-C3**	**Y1**	**Y2**	**Y3**	**Y4**	**GeoMean**	**Query/h**
SM	**gStore**	1,511	620	776	3,530	790	572	17,105	2,178	2,026	698
	RDF-3X	157	181	355	1,656	154	639,750	3,588	357	3,350	22
MR	**SHARD**	N/A	N/A	N/A	N/A	238,861	238,861	N/A	N/A	238,861	N/A
	H2RDF+	5,441	8,679	18,457	65,786	10,962	12,349	43,868	35,517	21,430	140
	CliqueSquare	29,216	23,908	40,464	55,835	139,021	73,011	36,006	100,015	77,755	41
	S2X	202,964	24,152	241,954	597,393	1,811,658	1,863,374	1,720,424	1,876,964	1,817,050	2
	S2RDF-VP	2,180	2,764	4,717	6,969	7,893	9,128	10,161	10,473	9,358	382
	S2RDF-ExtVP	1,880	2,269	3,531	5,295	2,822	3,032	3,393	3,628	3,203	1,118
SS	**AdPart-NA**	9	7	160	111	19	46	570	77	79	20,225
	TriAD	4	15	45	170	16	1,568	220	18	100	7,903
	Urika-GD	1,264	1,330	1,743	2,357	1,864	1,649	1,523	1,415	1,604	2,232
	H-RDF-3X	1,662	1,693	1,778	1,929	1,690	246,081	1,933	1,720	6,098	57
	SHAPE	1,870	1,824	1,836	2,723	1,824	665,514	1,823	1,871	8,022	21
	gStoreD	732	203	15,949	8,482	N/A	N/A	N/A	N/A	N/A	N/A
	DREAM (cached)	N/A	N/A	N/A	N/A	2,161	N/A	14,751	651	2,748	N/A

AdPart-NA and TriAD perform the best in the specialized systems category. They are comparable for L4 and L5 due to their high selectivity. AdPart-NA exploits its hash distribution to solve L4 and L5 without communication. It also solves L2 without communication but slower due to L2's low selectivity. AdPart-NA is more than an order of magnitude faster than TriAD and TriAD-SG for L2 because it reports the results through a single scan followed by hash lookups by utilizing hash indices and right deep tree plans. Compared to AdPart-NA, TriAD solves L2 by using two distributed index scans (one for each base subquery) followed by a binary search-based merge join that is only effective for selective queries. TriAD-SG outperforms TriAD and AdPart-NA in L6 as its pruning technique eliminates communication.

In DREAM, statistics are collected on a query-by-query basis and then cached for future queries. This causes huge performance difference between the first run of DREAM and its subsequent runs. The number of machines that can be utilized during query execution is bounded by the number of join vertices; only a maximum of 3 workers for LUBM queries. This explains its huge overhead when processing complex queries, i.e., L1 and L7. For L3, the query planner detects the empty result during statistics collection phase and terminates the query execution early. DREAM executes simple queries on a single machine by using RDF-3X.

4.4.2 WatDiv dataset

The WatDiv benchmark defines 20 query templates[1] classified into four categories: linear (L), star (S), snowflake (F) and complex queries (C). Table 7 shows the geometric mean of running 20 queries from each category on WatDiv-100M.

RDF-3X and gStore perform well for most queries, where they are faster than several distributed systems including S2X, CliqueSquare SHAPE and gStoreD. H2RDF+ and CliqueSquare perform worse than most distributed systems due to the overhead of MapReduce. H2RDF+ performs much better than CliqueSquare. Even though the flat plans of CliqueSquare reduce the number of MapReduce joins, H2RDF+ uses a more efficient join implementation using traditional RDF indices. Furthermore, H2RDF+ encodes the URIs and literals of RDF data; hence it has lower overhead than CliqueSquare when shuffling intermediate results. S2X performs much worse due to the significant network overhead of shuffling vertices with high memory footprint. The performance of S2RDF, on the other hand, is close to MapReduce-based systems on WatDiv dataset. Unlike the results in LUBM-10240, S2RDF-ExtVP shows only a slight improvement in comparison to S2RDF-VP.

Even though SHAPE and H-RDF-3X use 3-hop undirected guarantee replication, they do not outperform the single-machine RDF-3X. gStoreD is only faster than its centralized version (gStore) for L and S queries. Its parallelization overhead affected its performance on F and C queries. AdPart-NA and TriAD, on the other hand, provide better performance than all systems. TriAD performs better than AdPart-NA for L and F queries as it requires multiple subject-object joins. In contrast to AdPart, TriAD utilizes subject-object locality to answer these joins without communication. For complex queries with large diameters, AdPart-NA performs better as a result of its locality awareness. DREAM results are omitted because the overhead of its statistics calculation is extremely high (more than 24 hours) due to the high number of triple patterns in WatDiv.

4.4.3 YAGO dataset

YAGO2 does not provide benchmark queries. Therefore, we use the test queries (Y1-Y4) defined by AdPart to benchmark the performance of different systems (Table 7). Similar, to WatDiv dataset, H2RDF+ outperforms CliqueSquare and SHARD due to the utilization of HBase indices and its distributed implementation of merge and sort-merge joins. Moreover, S2X is still the slowest system while S2RDF provides better performance compared to MapReduce-based systems. Similar to the result in LUBM-10240, ExtVP shows better performance compared to VP. gStoreD fails to process all YAGO queries due to its parallelization overhead. AdPart-NA solves most of the joins in Y1 and Y2 without communication, which explains the superior performance of AdPart-NA compared to TriAD for Y1 and Y2, respectively. On the other hand, Y3 requires an object-object join on which AdPart-NA needs to broadcast the join column; therefore, it performs worse than TriAD.

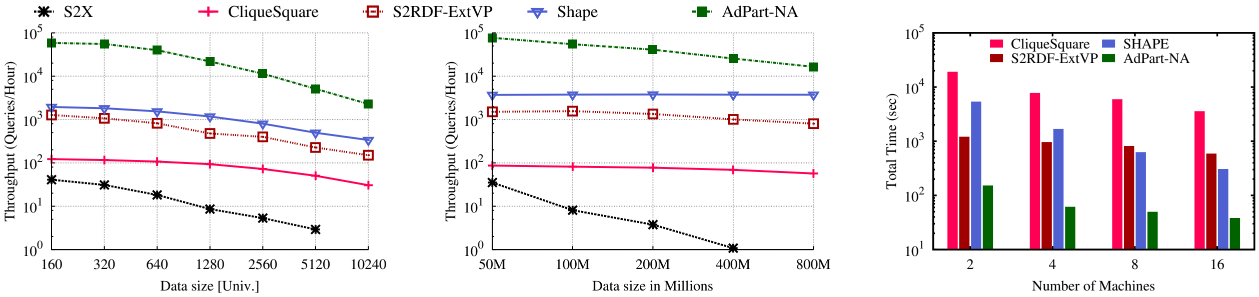
4.4.4 Bio2RDF dataset

Bio2RDF dataset does not have benchmark queries; therefore, we used the test queries (B1-B5) which are extracted from a real query log. The results of this experiment are shown in Table 8. Note that S2X failed to execute all Bio2RDF queries because of the huge amount of data it generates at its initial supersteps. Moreover, gStore, gStoreD, RDF-3X, H-RDF-3X, SHAPE, S2RDF and DREAM could not finish their preprocessing phase within 24 hours. The fastest systems in this experiment are AdPart-NA and TriAD while H2RDF+ and SHARD performed worse than other systems due to the MapReduce overhead. TriAD outperforms all other systems on queries B1, B2 and B3 since these queries require subject-object or object-object joins

[1] http://db.uwaterloo.ca/watdiv/basic-testing.shtml

Table 8: Query runtimes for Bio2RDF (ms). gStore, gStoreD, RDF-3X, H-RDF-3X, SHAPE, S2RDF and DREAM could not finish the data preprocessing within 24 hours while S2X failed to execute all queries.

	Bio2RDF	B1	B2	B3	B4	B5	GeoMean	Query/h
MR	SHARD	239,350	309,440	512,850	787,100	112,280	320,027	9
	H2RDF+	5,580	12,710	322,300	7,960	4,280	15,076	51
SS	AdPart-NA	17	16	32	89	1	15	116,129
	TriAD	4	4	5	N/A	2	4	N/A
	Urika-GD	879	798	1,832	1,180	947	1,075	3,194



(a) LUBM — Data Scalability (b) Watdiv — Data Scalability (c) LUBM-10240 — Strong Scalability

Figure 8: (a,b) Data Scalability, and (c) Strong Scalability

which contradict the initial partitioning of AdPart-NA. B4 is a complex query with 2-hop radius which AdPart-NA could efficiently process while TriAD crashed. B5 is a simple star query with only one triple pattern in which both AdPart-NA and TriAD exhibit comparable performance.

4.5 Scalability

In this section, we select a system from each category and test its data and strong scalability. More scalability experiments are available in [1]. We use AdPart-NA as a representative for specialized systems, SHAPE for systems that employ sophisticated partitioning, CliqueSquare for MapReduce-based, S2X for graph-based and S2RDF for sophisticated partitioning systems on Spark.

Data Scalability. In this experiment, we show how the performance of different distributed systems change as we increase the dataset size. Using LUBM, we generated 7 datasets ranging from 21 million triples to 1.3 billion triples; LUBM-160, LUBM-320, LUBM-640, LUBM-1280, LUBM-2560, LUBM-5120 and LUBM-10240. Similarly, we used WatDiv data generator to create 5 datasets with sizes ranging from 50 to 800 million triples. Figures 8(a) and 8(b) show the throughput (queries per hour) of the selected systems for both LUBM and WatDiv datasets. As expected, the throughput of most of the systems — excluding S2X — decrease slowly with the dataset size. AdPart-NA achieves the best throughput followed by SHAPE and S2RDF-ExtVP. Due to the scalability of MapReduce and Spark, CliqueSquare and S2RDF scale well with the data size. Similarly, SHAPE scales well as it does not incur communication during query evaluation. On the other hand, S2X does not scale because it generates a huge amount of intermediate results during BGP matching that excessively grows with larger datasets. Moreover, S2X could not evaluate LUBM-10240 and WatDiv-800M queries.

Strong Scalability. In this experiment, we use a workload of 35 queries to demonstrate the strong scalability of LUBM-10240 from 2 up to 16 machines. Strong Scalability for WatDiv-1B dataset is available in [17]. Figure 8(c) shows the time of each system as the number of machines increases. SHAPE achieves the best speedup as it solves queries without any communication overhead among machines. AdPart-NA incurs higher processing and communication overheads on larger clusters; therefore, it becomes less scalable after 4 machines. Finally, S2RDF-ExtVP does not scale well on larger cluster while CliqueSquare scales reasonably well as we increase the number of machines.

4.6 Workload Adaptivity

In this section, we evaluate the performance of workload-aware systems using SPARQL workloads on LUBM-10240 and WatDiv-1B. While Partout works for small datasets and workloads, it took a significant amount of time to preprocess the above datasets; it could not finish partitioning LUBM-10240 and WatDiv-1B within a day even for small workload sizes. Therefore, we only test the adaptivity of AdPart against two representative systems: AdPart-NA and S2RDF. AdPart-NA shows the best performance among specialized systems while S2RDF provides the best performance compared to other MapReduce-based systems.

LUBM-10240 workload: Using the 14 LUBM benchmark queries, a workload of 10K unique queries with different constants and structures is generated. Then, we shuffle the queries to generate a random workload for our experiments. This workload covers a broad spectrum of query complexities including simple selective queries, star queries as well as queries with complex structures and low selectivity.

WatDiv-1B workload: This workload contains a 5K-query from each query category (i.e., L, S, F and C) in WatDiv, resulting in a total of 20K queries. To simulate a change in the workload, queries of the same WatDiv-1B template are run consecutively. Therefore, after every sequence of 5K query executions, the type of queries changes.

Figure 9 shows the cumulative time as the execution progresses for both datasets. Without adaptivity, the cumulative time of both AdPart-NA and S2RDF increases sharply. AdPart-NA finished all LUBM-10240 queries under 1.5 hours while S2RDF could only finish 1460 queries within its 24 hours window (Figure 9(a)). Figure 9(b) also shows a similar behaviour where AdPart-NA finished all of the 20K queries of WatDiv-1B under 2 hours while S2RDF finished only 5100 queries. AdPart, on the other hand, adapts to the workload by redistributing frequently accessed data to allow future queries to be executed in parallel and without communication. Therefore, its performance becomes almost constant after 2.5K queries in LUBM-10240 allowing the workload to finish under 10 minutes. A similar behaviour is observed in WatDiv-1B, where the cumulative time becomes nearly constant after 10K queries and the workload completes in less than 15 minutes. In comparison to AdPart-NA, AdPart is an order of magnitude faster in the LUBM-10240 workload and 6 times faster in the WatDiv-1B workload.

4.7 Experiments Summary

4.7.1 Experiences with Evaluated Systems

AdPart-NA and TriAD: provide the lowest query runtime and startup overhead among all 12 distributed systems. However, they have huge memory overhead because they store the whole RDF graph in memory. For example, to query LUBM-10240, AdPart-NA and TriAD require 317GB and 232GB of RAM, respectively. AdPart is a good choice that reduces the end-to-end workload runtime by dynamically adapting its data distribution as the workload changes.

DREAM and gStoreD: are both disk-based systems that rely on existing single-machine engines; RDF-3X and gStore, respectively. DREAM replicates the full RDF graph to all workers, and incurs significant overhead for building and indexing the entire database on a single machine. We could not run gStoreD on LUBM-10240 and Bio2RDF as it requires significant time for data preprocessing.

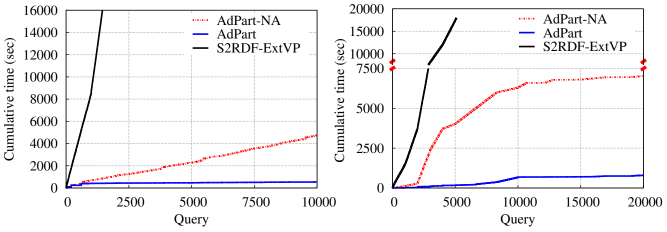
SHAPE and H-RDF-3X: significantly outperform MapReduce-based systems at the cost of higher preprocessing overheads. Semantic hash partitioning in SHAPE has a lower startup cost compared to METIS in H-RDF-3X. However, SHAPE only works for LUBM-10240 as it locates YAGO2 and WatDiv datasets in a single machine which results in a poor query performance.

SHARD, H2RDF+ and CliqueSquare: SHARD runs on all dataset but with slower query runtimes as it does not use sophisticated RDF indices or complex query optimizations. On the other hand, H2RDF+ uses specialized RDF indices and better-distributed join algorithms which allow it to significantly outperform SHARD. The optimizer of CliqueSquare uses its flat plans optimization to significantly reduce the join overhead of complex queries.

S2X and S2RDF: S2X has lightweight startup cost and huge runtime overhead caused by the size of intermediate results during BGP matching. As a result, it fails to run queries on LUBM-10240 and Bio2RDF. On the other hand, S2RDF has higher preprocessing overhead but shows significant performance improvement compared to MapReduce-based systems. This efficiency is mainly driven by its in-memory caching techniques and the materialization of different join reduction tables prior to query execution.

4.7.2 Discussion

Index Size. We noticed that the dataset size does not reflect the systems' actual memory/disk usage. Each system has its unique storage requirements which may exceed the original size of the input data. Typically, distributed systems load the RDF graph through complex data structures and sophisticated indexing; e.g. traditional SPO permutations. For example, to index WatDiv-1B (149GB), AdPart and TriAD require 208GB and 165GB, respectively. Details about the index sizes of various systems is available in [17].



(a) LUBM-10240 (b) Watdiv-1B

Figure 9: Cumulative execution time of a workload.

Influence of Query Complexity. Simple queries; e.g. LUBM (L4, L5 and L6) and WatDiv (star-shaped), require few joins or have a minimal number of results. Therefore, several systems within the same category perform comparably where even a centralized system may perform better than several distributed systems. As queries require a larger number of joins; e.g. LUBM (L1, L3 and L7) and WatDiv (snowflake and complex), or generate larger amounts of intermediate and final results; e.g. LUBM (L2 and L7), the advantage of distributed systems becomes visible. We noticed that the optimizations of each system favor certain query types. For example, CliqueSquare works well with complex queries where its flat plans significantly reduce the overhead of distributed joins. Likewise, AdPart works well when queries do not contain object-object joins that contradict its initial partitioning scheme.

Effect of Programming Language. We noticed that the programming language used in the evaluated systems does not significantly influence its performance. Rather, how each system parallelizes its tasks, reduces the communication cost and employs fewer synchronization barriers is more important. All specialized systems are built using C++; however, their diverging evaluation techniques result in various query performance. For example, while TriAD and AdPart-NA are both specialized in-memory systems, their performance is different because of their employed optimization techniques; e.g., intermediate result pinning vs. sharding and right-deep vs. bushy tree planning. The same applies to Java-based MapReduce systems.

5. CONCLUSION

In this paper, we provide an experimental evaluation of state-of-the-art distributed RDF systems. First, we classify and present a brief overview of systems in each category. Then using large-scale real and synthetic datasets, we extensively evaluated existing systems, through a wide range of SPARQL queries, considering different performance factors including startup overhead, incurred replication, query performance, and scalability.

We highlight the following limitations and possible future research directions for optimizing distributed RDF systems: (*i*) all existing systems focus exclusively on supporting BGP SPARQL queries. Hence, they do not support evaluating important generic operators such as FILTER, LIMIT and OPTIONAL. (*ii*) Several systems are highly optimized for specific datasets, e.g., LUBM, and do not perform well for other datasets. (*iii*) Most distributed systems do not support updates, adding new triples would require several systems to rebuild their indices, which is very expensive for large-scale graphs. Finally, (*iv*) no partitioning algorithm suits all RDF graphs and all query types. The complexity of sophisticated partitioning schemes does not allow distributed RDF systems to process very large graphs in a timely manner. At the same time, they do not always guarantee better performance compared to lightweight partitioning. We believe that focusing on the problem of matching partitioning strategies to

RDF graphs, instead of using a single partitioning strategy, is a promising research direction.

6. REFERENCES

[1] https://github.com/ecrc/rdf-exp.

[2] Apache Hadoop. http://hadoop.apache.org/.

[3] Bio2RDF. http://bio2rdf.org/.

[4] Dbpedia. http://dbpedia.org/.

[5] HBase. http://hbase.apache.org.

[6] LUBM. http://swat.cse.lehigh.edu/projects/lubm/.

[7] PubChemRDF. http://pubchem.ncbi.nlm.nih.gov/rdf/.

[8] RDF Primer. http://www.w3.org/TR/rdf-primer/.

[9] SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/.

[10] UniProt. http://www.uniprot.org/.

[11] Urika-GD. http://www.cray.com/sites/default/files/resources/Urika-GD-TechSpecs.pdf.

[12] WatDiv. http://db.uwaterloo.ca/watdiv/.

[13] YAGO2. http://yago-knowledge.org/.

[14] A. Schätzle, M. Zablocki and G. Lausen. PigSPARQL: Mapping Sparql to Pig Latin. In *Proc. of SWIM*, 2011.

[15] A. Schätzle, M. Zablocki, S. Skilevic and G. Lausen. S2RDF: RDF Querying with SPARQL on Spark. *PVLDB*, 9(10):804–815, 2016.

[16] I. Abdelaziz, M. R. Al-Harbi, S. Salihoglu, and P. Kalnis. Combining Vertex-centric Graph Processing with SPARQL for Large-scale RDF Data Analytics. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2017.

[17] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data: Technical Report. https://github.com/ecrc/rdf-exp/raw/master/TRv1.pdf.

[18] I. Abdelaziz, R. Harbi, S. Salihoglu, P. Kalnis, and N. Mamoulis. Spartex: a Vertex-centric Framework for RDF Data Analytics. *PVLDB*, 8(12):1880–1883, 2015.

[19] M. Armbrust, R. S. Xin, Lian, et al. Spark Sql: Relational Data Processing in Spark. In *Proc. of SIGMOD*, 2015.

[20] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" Loaded: a Scalable Lightweight Join Query Processor for RDF Data. In *Proc. of WWW*, 2010.

[21] B. Shao, H. Wang and Y. Li. Trinity: a Distributed Graph Engine on a Memory Cloud. In *Proc. of SIGMOD*, 2013.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins. Pig Latin: a Not-so-foreign Language for Data Processing. In *Proc. of SIGMOD*, 2008.

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

[24] D. Abadi, A. Marcus, S. Madden and K. Hollenbach. SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. *The VLDB Journal*, 18(2):385–406, 2009.

[25] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz and S. Zampetakis. CliqueSquare: Flat Plans for Massively Parallel RDF Queries. In *Proc. of ICDE*, 2015.

[26] D. Faye, O. Cure, and D. Blin. A Survey of RDF Storage Approaches. *ARIMA Journal*, 15:11–35, 2012.

[27] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[28] Hose, K. and Schenkel, R. WARP: Workload-aware Replication and Partitioning for RDF. In *Proc. of ICDE Workshops*, 2013.

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

[30] M. Husain, J. McGlothlin, M. Masud, L. Khan, and B. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *TKDE*, 23(9), 2011.

[31] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of OSDI*, 2004.

[32] K. Lee and L. Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB*, 6(14), 2013.

[33] K. Zeng, J. Yang, H. Wang, B. Shao and Z. Wang. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB*, 6(4), 2013.

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

[35] M. Kornacker, A. Behm, V. Bittorf, et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proc. of CIDR*, 2015.

[36] L. Galarraga, K. Hose and R. Schenkel. Partout: A Distributed Engine for Efficient RDF Processing. *CoRR*, abs/1212.5636, 2012.

[37] L. Zou, J. Mo, L. Chen, M.T. Özsu and D. Zhao. gStore: Answering SPARQL Queries via Subgraph Matching. *PVLDB*, 4(8):482–493, 2011.

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

[39] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker and I. Stoica. Spark: Cluster Computing with Working Sets. *Proc. of HotCloud*, 10, 2010.

[40] Z. Ma, M. A. Capretz, and L. Yan. Storing Massive Resource Description Framework (RDF) data: a Survey. *The Knowledge Engineering Review*, 31(4):391–413, 2016.

[41] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras and N. Koziris. H2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs. In *Big Data*, 2013.

[42] O. Hartig and M. T. Özsu. Linked Data Query Processing. In *Proc. of ICDE*, 2014.

[43] M. T. Özsu. A survey of RDF Data Management Systems. *Frontiers of Computer Science*, 10(3):418–432, 2016.

[44] P.A. Bernstein and D.W. Chiu. Using Semi-Joins to Solve Relational Queries. *Journal of the ACM*, 28(1):25–40, 1981.

[45] P. Peng, L. Zou, M. T. Özsu, L. Chen, and D. Zhao. Processing SPARQL Queries over Distributed RDF Graphs. *The VLDB Journal*, 25(2):243–268, 2016.

[46] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim and M. Sahli. Accelerating SPARQL Queries by Exploiting Hash-based Locality and Adaptive Partitioning. *The VLDB Journal*, 25(3):355–380, 2016.

[47] K. Rohloff and R. E. Schantz. Clause-iteration with MapReduce to Scalably Query Datagraphs in the SHARD Graph-store. In *Proc. of DIDC*. ACM, 2011.

[48] S. Gurajada, S. Seufert, I. Miliaraki and M. Theobald. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *Proc. of SIGMOD*, 2014.

[49] S. Sakr and G. Al-Naymat. Relational Processing of RDF Queries: a Survey. *SIGMOD Record*, 38(4):23–28, 2010.

[50] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen. Sempala: Interactive SPARQL Query Processing on Hadoop. In *Proc. of ISWC*, 2014.

[51] A. Schätzle, M. Zablocki, T. Berberich, and G. Lausen. S2X: Graph-parallel Querying of RDF With GraphX. In *VLDB Workshop on Big Graphs Online Querying*, 2015.

[52] M. Svoboda and I. Mlýnková. Linked Data Indexing Methods: a Survey. In *OTM*, pages 474–483, 2011.

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

[54] K. Wilkinson. Jena Property Table Implementation. In *SSWS*, pages 35–46, 2006.

[55] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proc. of SWDB*, pages 131–150, 2003.

[56] Xiaofei Zhang and Lei Chen and Yongxin Tong and Min Wang. EAGRE: Towards Scalable I/O Efficient SPARQL Query Evaluation on the Cloud. In *Proc. of ICDE*, 2013.

[57] S. Yang, X. Yan, B. Zong, and A. Khan. Towards Effective Partition Management for Large Graphs. In *Proc. of SIGMOD*, 2012.