Rainbow: A Distributed and Hierarchical RDF Triple Store with Dynamic Scalability

Rong Gu, Wei Hu, Yihua Huang
National Key Laboratory for Novel Software Technology
Nanjing University, Nanjing, China 210093
Email: gurong@smail.nju.edu.cn, {whu, yhuang}@nju.edu.cn

***Abstract*—In the Big Data era, the ever-increasing RDF data have reached a scale in billions of triples and brought obstacles and challenges to single-node RDF data stores. As a result, many distributed RDF stores have been emerging in the Semantic Web community recently. However, currently published ones are either not enough efficient on performance or failed to achieve flexible scalability. In this paper, we propose Rainbow, a scalable and efficient RDF triple store. The RDF data indexing scheme in Rainbow is a hybrid one which is designed based on the statistical analysis of user query space. Further, to better support the hybrid indexing scheme, Rainbow adopts a distributed and hierarchical storage architecture that uses HBase as the scalable persistent storage and combines a distributed memory storage to speedup query performance. The RDF data in memory storage is partitioned by the consistent hashing algorithm to achieve the dynamic s-calability. Experiments show that Rainbow outperforms typical existing distributed RDF triple stores, with excellent scalability and fault tolerance.**

***Keywords*-SPARQL; RDF; big data; distributed computing**

I. Introduction

In the Big Data era, the number of RDF triples in large public knowledge bases, e.g., DBpedia and Bio2RDF, has increased to billions. Although the increasing amount of RDF data is good for the Semantic Web initiative, it also brings performance bottlenecks to the current RDF data management systems. There is significant progress made by the research community towards efficient storage and query of large-scale RDF data sets [1], [2], [3], [4], [5]. Generally speaking, these distributed RDF stores can be classified into two categories [6]. One is to develop highly scalable distributed systems for storing and querying large collections of RDF triples, called *triple stores*. They usually focus on taking advantages of Apache Hadoop, HBase or Pig to attain the scalability, e.g., Jena-HBase [3], SHARD [4] and PigSPARQL [5]. The other type of methods refers to *graph partition stores* [2], and leverages graph partitioning optimization techniques to improve RDF graph pattern matching efficiency. One bottleneck for the graph partition stores is, when new batches of data or machines added, the whole RDF graph usually needs to be repartitioned for load balance, which is a time-consuming process and even requires to interrupt the query service.

In this paper, we propose *Rainbow*, a distributed and efficient RDF triple store. The major contributions of our work are summarized as follows:

- By analyzing real-user and synthetic benchmark queries, we proposed a hybrid indexing scheme for efficiently storing and querying large-scale RDF data.
- To well support the proposed hybrid indexing scheme, we introduce a distributed and hierarchical storage architecture. Besides adopting a distributed storage for data persistence, we also build a distributed memory storage to store frequently-used RDF data indices for fast random access.
- Rainbow is dynamically scalable. We leverage the consistent hashing algorithm to partition data across distributed memory storage, which allows adding or removing memory storage resources, when needed, without totally repartitioning the indices.
- A fault-tolerant mechanism is also designed to assure the successful query response in Rainbow .

II. Related Work

Many research efforts have been devoted to develop distributed RDF data management systems. To achieve scalability, some systems are implemented on the Hadoop computing framework. For instance, SHARD [4] stores RDF triples in flat text files in HDFS and matches triple patterns with MapReduce. PigSPARQL [5] translates SPARQL queries to Pig Latin programs and executes them by MapReduce jobs. SPIDER [1] stores RDF data in HBase but still suffers from the high latency of MapReduce jobs. It is believed that MapReduce is more suitable for offline batch processing than online services. Vaibhav et al. [3] proposed Jena-HBase, which stores RDF indices in HBase and directly carries out queries through HBase APIs.

Some other researchers use graph partitioning optimization techniques for RDF graph pattern matching. For example, Huang et al. [2] reported that their graph partition store achieved high query performance over RDF graphs. It works well for the case in which RDF graphs are static, and machine number is pre-fixed. However, when new batches of data or machines added, the whole RDF graph needs to be repartitioned and reassigned for better load balance. Such

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

indexing schemes are not always dynamically salable for both input data and underlying systems. Thus, they are not suitable for many applications with streaming or incremental input data, e.g., social networks, because they contain fast evolving entities, and the graph structures keep changing.

III. Indexing and Data Placement

A. Query Space Analysis

In triple stores, indices are usually built for improving the querying efficiency of the triple patterns. Many existing RDF triple stores design indexing schemes from the perspective of database which focus on the structures of RDF data. However, from the perspective of information retrieval, these methods only study the document space but neglect the query space. Studying user queries is helpful for designing efficient indexing scheme for real-world applications.

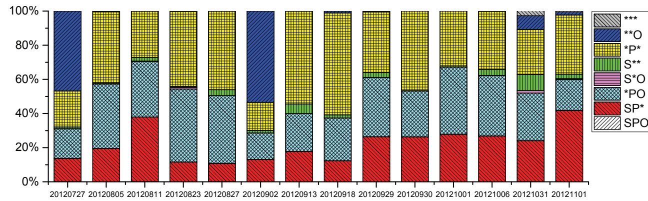

Figure 1. Statistics of real-user query patterns (extracted from the whole DBpedia query logs)

We analyze millions of real user queries[1] and synthetic benchmark(LUBM, LongWell, SP2Bench etc.) queries.The statistical results are shown in Fig. 1. It can be observed that triple patterns SP*, *PO and *P* are always more frequently issued in real user queries than the others. The total proportion of these three triple patterns also reaches to ∼90% in LUBM, LongWell, SP2Bench and Berlin SPARQL Queries (the detailed results are omitted due to space limitation). This inspires us to build efficient indices on these three triple patterns to accelerate query speed, which would benefit a bulk of SPARQL queries.

B. Indexing Scheme

Based on the statistical analysis above, we design a hybrid RDF data indexing scheme with its data placement policy that well fit our distributed hierarchical storage architecture. As shown in Fig. 2, the indices are maintained in hash tables. The SP*, *PO triple pattern indices are more desirable to be resided in distributed memory. The *P* can be extend to SP* and *PO.

Besides indexing the frequently queried triple patterns in the memory storage, it is also needed to build a general indexing scheme on the persistent storage. By investigating current RDF indices on HBase, the *Three Table Index* put forward in [7] is adopted for its compactness and generality.

[1]around 22 million valid queries downloaded from ftp://download.openlinksw.com/support/dbpedia/

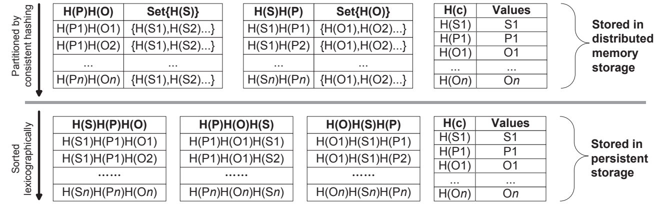

Figure 2. The hybrid indexing scheme and data placement of RDF triples

The three index tables are named SPO, POS, and OSP. Each triple compacts its components by each table name's order and stored in each table's row key. As row keys in HBase tables are indexed and sorted lexicographically, the *Three Table Index* is sufficient to answer each possible triple pattern by using only a range scan [7].

Similar to some RDF triple stores, S, P and O are encoded in our storage (MD5 hash coding is adopt here). By encoding, the storage space of RDF triples is compressed as compared with long literals or URIs. This improves the space utilization of memory and disk storage and reduces the size of intermediate data. Since the intermediate data need to be transferred over network during querying, transferring less data would lead to query performance improvement.

IV. System Architecture

In this section, we begin with an overview of the Rainbow architecture, and then give detailed description of its important modules and features, such as distributed hierarchical storage, fault tolerance, scalability and elasticity.

A. Overview

Fig. 3 shows the main components of Rainbow lie in three layers. The bottom layer is a distributed persistent storage. In our implementation, HBase is adopted. The middle layer is the distributed memory storage layer, consisting of a set of single-node in-memory stores. On the top lies the access layer providing data loading and query services to end users. A daemon, called *client* hangs there, answers users' queries. It has three important modules. The first one is a *SPARQL engine*, responsible for parsing SPARQL queries and generating query plans with optimization. The second is named *store selection*, which decides to perform queries on which in-memory store, or switch to the underlying HBase. The third is called *store connection*, which manages the storage connections and monitors the states of in-memory stores from ZooKeeper. This can reduce the store connection overhead during querying and recognize in-memory store failures. The *data loader* takes charge of importing RDF triples.

B. Distributed Hierarchical Storage Module

To better support the hybrid indexing scheme put forward in Section III, Rainbow is built on a distributed, hierarchical

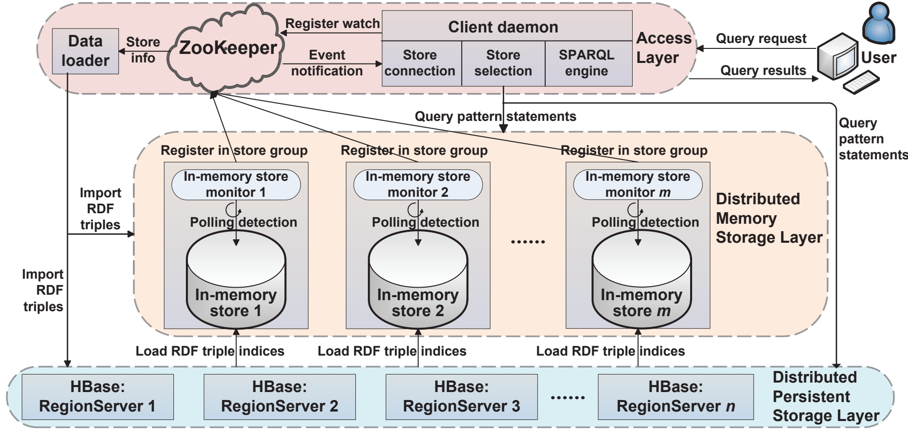

Figure 3. An overview of Rainbow's system architecture

storage. On one hand, the underlying storage persists data across machines' disks with multi-replicates. It has large capacity and high reliability, but is inefficient for random data access. On the other hand, the distributed memory storage resides all data in memory. It has less capacity but highly random data access speed. The data loader imports RDF triples into both the distributed persistent storage and the distributed memory storage simultaneously. Moreover, the frequently queried RDF triples would be migrated to the distributed memory storage from the underlying persistent storage. During loading or migration, the quite unused data would be replaced by new-coming data according to the least recently used (LRU) policy if data resided in memory exceeds quota. Many distributed RDF triple stores passively use memory through the OS's caching mechanism [2], [3], which is implicit and coarse-grained. But our in-memory stores allow to explicitly specify which RDF data resides in memory in a fine grain way. This feature well supports the hybrid indexing scheme described in Section III.

Many distributed RDF triples stores [1], [3] report HBase is scalable and fault-tolerant. In our implementation, HBase is chosen as our distributed persistent storage. For the memory storage, Rainbow adopts Redis as each single-node in-memory store. The whole distributed memory storage is made up of all these Redis stores to provide a unified interface to applications. Data stored in the Redis stores are partitioned by the consistent hashing algorithm with good scalability and load balance [8].

C. Fault Tolerance Module

Fault tolerance and high availability are important for real-world applications. The persistent storage of Rainbow inherits the fault tolerance feature from HDFS/HBase, which achieves fault tolerance by data replication.

However, the distributed memory storage is made by a batch of independent single-node in-memory stores. Therefore, it needs to deal with fault tolerance by itself. Replicating data across in-memory stores decreases the limited capacity of memory storage. Instead, we focus on detecting failures of in-memory stores. After the failures are detected, the corresponding data is migrated from the persistent storage to live in-memory stores in a lazy way. The failure detection mechanism leverages the *Configuration Management* and *Group Membership* services in ZooKeeper.

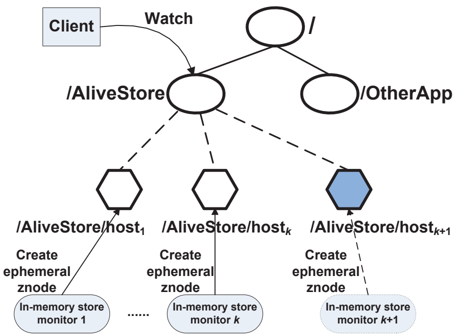

Figure 4. Fault detection mechanism in our distributed memory storage (oval denotes regular znode and hexagon denotes ephemeral znode, they are all kept in ZooKeeper)

As presented in Fig. 4, there is a daemon running on both the client machine and each in-memory store machine. The $/AliveStore$ directory in ZooKeeper keeps the group membership information of the in-memory stores. The client daemon keeps a *Watch* on this directory, and the *in-memory store monitor* daemon modifies the child of $/AliveStore$ in ZooKeeper when the in-memory store's status changes. In this way, the client can always recognize which in-memory stores are alive and which are dead. When needing to access data on a failed in-memory store, the system can automatically switch to the persistent storage layer for the data. Then, the data from the persistent storage layer

would be loaded to live in-memory stores according to the consistent hashing policy (see subsection IV-D).

D. Scalability and Elasticity

Rainbow's distributed memory storage adopts consistent hashing data partition algorithm [8] to achieve scalability. The basic idea of consistent hashing is decoupling the value of a key from the host that it is stored on. The space of hash keys are arranged into a ring. Both hosts and data are mapped on the ring by hashing. Each host in the ring is responsible for keys that lie in the range from its token to the nearest following token. New added hosts would be allocated to a part of non-overlapped key space. From the perspective of storage, this shared-nothing architecture can almost scale without limit [9]. The distributed persistent storage in Rainbow achieves the scalability from HBase.

V. Query Processing

In this section, we introduce the query processing mechanism in Rainbow. The storage architecture of Rainbow is independent to the choice of SPARQL query processing engines. In our implementation, Sesame [10], a well-known framework for processing RDF data, is adopted. We implement the customized SAIL APIs provided by Sesame to conduct RDF queries using Rainbow as the backend RDF storage. The entire query processing workflow is outlined in Fig. 5.

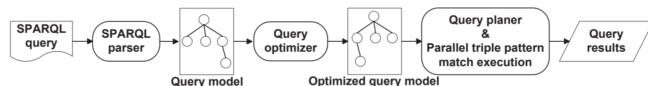

Figure 5. SPARQL query processing workflow

The SPARQL parser accepts a SPARQL query as input and generates its corresponding query model m, which consists of a batch of triple pattern match and join operations. The execution order of triple patterns in a SPARQL query significantly affects its performance [7], [11]. The query optimizer applies its optimization strategies to m and then generates an optimized query model m'. Currently, Rainbow integrates two optimization rules to ensure the execution of triple patterns with high selectivity prior to those with low selectivity. It is worth noting that our optimizer module is extensible. If desirable in future, it is easy to add more heuristic rules into it.

- **Rule 1.** Intuitively, triple patterns with more binding values are likely to have less matches. Thus, we set higher execution priority for the triple patterns with more binding values than variables. This heuristic rule works well in our scenario.
- **Rule 2.** For those triple patterns having same variables, we adopt an optimization rule with statistics [7]. In Rainbow, a metadata table resides in the client's memory. It records the occurrence frequency of indexing keys in raw data and whether they have been stored in the distributed memory storage. This metadata is useful to estimate the selectivity and query performance of triple patterns. The metadata table is updated when new RDF data are inserted. If allowed, the update can also be performed in batch.

After the optimized query model m' is generated, a series of single triple pattern match operations are executed. Besides the default streaming mode in Sesame, we also implement a parallel mode for improving efficiency by better utilizing the underlying distributed architecture. In the parallel mode, triple patterns are sent to the corresponding storage nodes and processed in parallel. Meanwhile, for a triple pattern, its results are returned in the batch style with high throughput rather than one by one that incurs overhead of too many RPC connections. Each single triple pattern match execution needs to find a proper storage layer according to its type. The single triple pattern matching algorithm is described Algorithm 1.

Algorithm 1: Single triple pattern matching

```
  Input: An input triple pattern, denoted by tp
  Output: The set of RDF triples that match tp, denoted by results
1 begin
2   if tp.type ∈ {SP*, *PO} and ExistsInMemory(tp) then
3     results ← QueryInMemory(tp);
4   else
5     results ← QueryOnHBase(tp);   /* Retrieve from
      HBase */
6     ... /* Start a thread to load and convert
        the RDF data indices into memory by
        the consistent hashing algorithm     */
7   end
8   return results;
9 end
```

VI. Evaluation

In this section, we evaluate the performance of Rainbow. For comparison, we also verify the performance of several typical distributed RDF triple stores under the same environment.

A. Experiment Setup

We conducted experiments on a local cluster with 18 computing nodes. Each node has two Xeon Quad 2.4 GHz processors, 24 GB memory and two 2 TB 7200 RPM SATA hard disks. The nodes are connected with 1 Gb/s Ethernet. LUBM (the Lehigh University Benchmark) was used as the benchmark for test. In our experiments, three scale-level datasets were generated with 10, 100 and 1000 universities, containing around 1.3 million, 13.6 million and 136 million triples, respectively.

1) Query Performance Comparison: For simplicity, the distributed memory storage in Rainbow is denoted as *Rainbow-IM*, and the underlying distributed persistent storage is denoted as *Rainbow-HBase*. Their performance are listed separately. All the time presented here is the average of 10 runs of the queries. The detailed results are shown in Table I–III.

We denoted the cases failed to return answers in a reasonable time (1000s here) as *abort*. Jena-HBase failed to execute all the queries on LUBM-100 and LUBM-1000. However, from LUBM-10, it can be observed that the performance of Rainbow outperforms Jena-HBase on all the queries. The experiments in [3] also reflected that even for the fast queries, the execution time of Jena-HBase increased linearly as the size of a dataset increases. On the contrary, the fast queries performance of Rainbow is stable due to its selective indexing scheme and query optimization strategies. Rainbow also outperforms SHARD three orders of magnitude for most cases. This is because SHARD converted each query to a series of MapReduce jobs that incur job initialization overheads and a large amount of over network disk reads and writes.

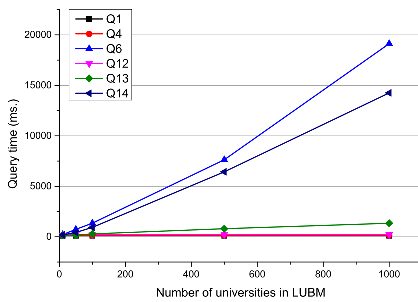

Figure 6. Varying the amount of data

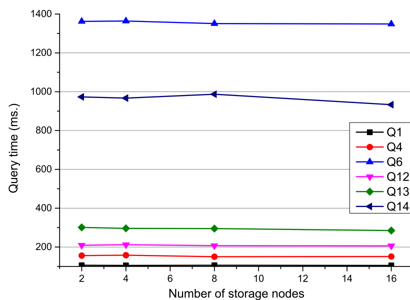

Figure 7. Data and machine scalability of Rainbow-IM

2) Scalability: Varying the amount of data. We tested Rainbow-IM on a 17-node cluster with 5 LUBM datasets from 10 universities to 1000. The results are shown in Fig. 6. For the fast queries like Q1, Rainbow-IM achieves the constant size of intermediate results and stable performance regardless of the data size. For the slow queries like Q6, Rainbow-IM scales linearly as the size of the dataset increases.

Varying the number of machines. We evaluated Rainbow-IM on LUBM-100 by varying the number of machines in a cluster. The results are shown in Fig. 7. For all the queries, Rainbow-IM achieves stable performance regardless of increasing or decreasing the number of machines. This is because the index data can already be kept in the memory with few machines. Adding machines into Rainbow-IM can enlarge the storage capacity of the distributed memory, which can potentially achieve more performance improvement when more data comes to the system.

3) Dynamic and Fault Tolerant Features: In addition, we designed a simulating experiment to verify the dynamic scalability and fault tolerance features of Rainbow. The scenario is designed as follow: a user submits a sequence of queries to a running Rainbow.

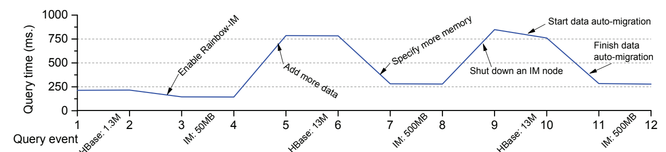

Figure 8. Dynamic scalability and fault tolerance of Rainbow with simulated events

Initially, Rainbow-HBase held 1.3M triples while Rainbow-IM was disabled. Then, a sequence of simulated query events was triggered (see Fig. 8), and the query performance varied with these events. Queries were executed by switching to Rainbow-HBase when Rainbow meets shortage of memory (Query events 5 and 6) or machine failure (Query events 9 and 10). After memory dynamically added (Query events 3, 4, 7 and 8), or the required data migrated to live in-memory stores Query events 11, 12), the performance was improved on the fly.

VII. Conclusion and Future Work

In this paper , we introduced Rainbow which a scalable and efficient RDF triple store. The storage of Rainbow is distributed and hierarchical and it resides frequently-used RDF data indices in distributed memory for fast random access. Based on the statistical analysis of large query space, Rainbow designs a hybrid indexing scheme that well fits its storage architecture. Furthermore, Rainbow also provides dynamic scalability and fault-tolerance mechanisms.

In future work, we will study on adding new storage media such as solid state disk (SSD) into our hierarchical architecture for further performance improvement. Also, we look forward to tuning key parameters in our query optimization with machine learning methods.

Table I
QUERY EXECUTION TIME ON LUBM-10 (1.3 MILLION TRIPLES)

Unit: ms.	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
Rainbow-IM (Cold)	222	9,766	212	362	367	482	339	1,352	11K	220	313	328	319	437
Rainbow-IM (Hot)	**104**	9,257	**106**	**148**	**143**	**213**	**124**	**526**	11K	**108**	**132**	**175**	**143**	**185**
Rainbow-HBase (Cold)	140	25K	196	431	855	37K	181	23K	25K	124	375	1,919	256	30K
Rainbow-HBase (Hot)	121	23K	121	344	583	28K	134	22K	25K	118	221	1,313	226	23K
Jena-HBase (Cold)	22K	abort	62K	4,326	59K	1,442	abort	abort	abort	15K	4,444	89K	243K	1,015
Jena-HBase (Hot)	14K	abort	43K	3,712	8,861	238	abort	288K	abort	15K	1,600	85K	181K	597
SHARD (Cold)	153K	273K	150K	360K	129K	54K	192K	340K	430K	175K	149K	290K	111K	40K
SHARD (Hot)	105K	244K	105K	201K	99K	34K	165K	230K	250K	106K	101K	196K	110K	33K

Table II
QUERY EXECUTION TIME ON LUBM-100 (13.6 MILLION TRIPLES)

Unit: ms.	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
Rainbow-IM (Cold)	223	abort	220	278	381	1,844	326	1,375	11K	223	311	433	650	1,302
Rainbow-IM (Hot)	**105**	abort	**106**	**151**	**144**	**1,349**	**124**	**635**	11K	**108**	**138**	**206**	**285**	**933**
Rainbow-HBase (Cold)	230	abort	225	452	582	800K	204	28K	31K	128	234	2,103	937	301K
Rainbow-HBase (Hot)	126	abort	124	415	512	363K	198	26K	30K	125	213	1,807	807	270K
Jena-HBase	abort	abort	abort	abort	abort	abort	abort	abort	abort	abort	abort	abort	abort	abort
SHARD (Cold)	111K	367K	134K	321K	126K	91K	214K	363K	366K	146K	153K	285K	129K	82K
SHARD (Hot)	110K	281K	133K	229K	112K	39K	195K	283K	272K	109K	107K	196K	111K	39K

Table III
QUERY EXECUTION TIME ON LUBM-1000 (136 MILLION TRIPLES)

Unit: ms.	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
Rainbow-IM (Cold)	220	abort	226	286	274	21K	227	1,639	12K	243	275	220	1,694	18K
Rainbow-IM (Hot)	**107**	abort	**107**	**151**	**156**	**19K**	**125**	**654**	12K	**108**	**139**	**219**	**1,343**	**14K**
Rainbow-HBase (Cold)	165	abort	181	834	1,960	abort	503	35K	36K	161	758	7,831	8,799	abort
Rainbow-HBase (Hot)	145	abort	150	732	728	abort	279	26K	33K	151	659	2,050	4,816	abort
Jena-HBase	abort	abort	abort	abort	abort	abort	abort	abort	abort	abort	abort	abort	abort	abort
SHARD (Cold)	324K	925K	318K	681K	311K	130K	574K	643K	868K	348K	288K	511K	270K	150K
SHARD (Hot)	268K	**741K**	278K	598K	266K	114K	514K	605K	749K	265K	259K	490K	259K	114K

ACKNOWLEDGMENT

This work is funded in part by China NSF Grants (No. 61223003 and 61370019) and the USA Intel Labs University Research Program.

REFERENCES

[1] C. Y. e. a. Choi H., Son J., "Spider: A system for scalable, parallel/distributed evaluation of large-scale rdf data," in *Proc. 18th ACM Conference on Information and Knowledge Management*. ACM, 2009, pp. 2087–2088.

[2] R. K. Huang J., Abadi D.J., "Scalable sparql querying of large rdf graphs," *Proc. of the VLDB Endowment*, vol. 4, no. 11, pp. 1123–1134, 2011.

[3] T. B. e. a. Khadilkar V., Kantarcioglu M., "Jena-hbase: A distributed, scalable and efficient rdf triple store," in *Proc. Int. Semantic Web Conference(ISWC)*. Springer, 2012, pp. 1–4.

[4] S. R. Rohloff K., "High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store," in *Proc. Programming Support Innovations for Emerging Distributed Applications*. ACM, 2010, pp. 4:1–4:5.

[5] L. G. Scätzle A., Przyjaciel-Zablocki M., "Pigsparql: Mapping sparql to pig latin," in *Proc. the 2013 ACM Special Interest Group on Management Of Data(SIGMOD)*. ACM, 2013, pp. 505–516.

[6] C. A. e. a. Kawa A., Bolikowski Ł., "Data model for analysis of scholarly documents in the mapreduce paradigm," in *Proc. Intelligent Tools for Building a Scientific Information Platform*. Springer, 2013, pp. 155–169.

[7] R. D. Punnoose R., Crainiceanu A., "Rya: A scalable rdf triple store for the clouds," in *Proc. 1st International Workshop on Cloud Intelligence*. ACM, 2012, pp. 4:1–4:8.

[8] L. T. e. a. Karger D., Lehman E., "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. 29th Annual ACM Symposium on Theory of Computing*. ACM, 1997, pp. 654–663.

[9] L. Y. Shao B., Wang H., "Trinity: A distributed graph engine on a memory cloud," in *Proc. 3rd International Workshop on Semantic Web Information Management*. ACM, 2011, pp. 4:1–4:8.

[10] V. H. F. Broekstra J., Kampman A., "Sesame: A generic architecture for storing and querying rdf and rdf schema," in *Proc. Int. Semantic Web Conference(ISWC)*. Springer, 2002, pp. 54–68.

[11] W. H. e. a. Zeng K., Yang J., "A distributed graph engine for web scale rdf data," *Proc. of the VLDB Endowment*, vol. 6, no. 4, pp. 265–276, 2013.