ScalaRDF: a Distributed, Elastic and Scalable In-Memory RDF Triple Store

Chunming Hu, Xixu Wang, Renyu Yang, Tianyu Wo
State Key Laboratory of Software Development Environment (NLSDE)
School of Computer Science and Engineering
Beihang University, Beijing, China
{*hucm, wangxx, yangry, woty*}*@act.buaa.edu.cn*

***Abstract*—The Resource Description Framework (RDF) and SPARQL query language are gaining increasing popularity and acceptance. The ever-increasing RDF data has reached a billion scale of triples, resulting in the proliferation of distributed RDF store systems within the Semantic Web community. However, the elasticity and performance issues are still far from settled in face of data volume explosion and workload spike. In addition, providers face great pressures to provision uninterrupted reliable storage service whilst reducing the operational costs due to a variety of system failures. Therefore, how to efficiently realize system fault tolerance remains an intractable problem. In this paper, we introduce ScalaRDF, a distributed and elastic in-memory RDF triple store to provision a fault-tolerant and scalable RDF store and query mechanism. Specifically, we describe a consistent hashing protocol that optimizes the RDF data placement, data operations (especially for online RDF triple update operations) and achieves an autonomously elastic data re-distribution in the event of cluster node joining or departing, avoiding the holistic oscillation of data storage. In addition, the data store is able to realize a rapid and transparent failover through replication mechanism which stores in-memory data replica in the next hash hop. The experiments demonstrate that query time and update time are reduced by 87% and 90% respectively compared to other approaches. For an 18G source dataset, the data redistribution takes at most 60 seconds when system scales out and at most 100 seconds for recovery when nodes undergo crash-stop failures.**

Keywords*—RDF; distributed system; consistent hashing protocol; scalability;

I. INTRODUCTION

The Resource Description Framework (RDF) is proliferating among a variety of fields, such as science, bioinformatics, social networks, knowledge graphs, and auto question&answer services. For instance, semantic-web style ontologies and knowledge bases with millions of facts from DBpedia [6], Probase [23], Wikipedia [17] and Science Commons [27] are now publicly available. Many industrial search engines from Google, Bing and Yahoo! [14] [29] have presented substantial supports for RDF to explicitly express the semantics of their web contents. In reality, RDF is designed to flexibly model the schema-free information for the Sematic Web [17] [27] and it forms the data items as triples. All RDF stores can be searched by using SPARQL query language that is mainly composed of triple patterns.

The rapid explosion of data volume and dynamicity urgently necessitates an elastic and scalable data provisioning and storage. The dynamicity stems from either the fluctuation of data scale and request bursting due to world-wide hot events or updates of the web contents within Internet environments [25]. Obviously, the bottleneck of centralized RDF systems [3] [7] [17] [22] [27] is subject to scalability and reliability limitations derived from the restricted computational capacities and storage resources of a single machine. Due to this reason, many distributed RDF store and query approaches are proposed to improve the system scalability and accelerate the query performance.

In reality, the RDF can be inherently expressed as graph and the graph partition can be accordingly utilized to decide the placement pattern and storage position. However, the graph-based systems have to perform a holistic data re-partition among existing machine nodes for load-balancing on the occasion of new data storing or machines joining-in. This process of data re-distribution is extremely time-consuming considering the massive data amount and the varying condition in web-scale environments. Query services even have to endure service downtimes for the upgrading of underlying storage systems, resulting in significant performance degradation. To overcome these efficiency issues, in-memory relational data table schemes can facilitate the reduction of such operational costs. Besides, dynamic update of RDF data source is significantly important since knowledge-base systems [6] [17] [23] allow for online editing and data sharing among users. The update frequency will dramatically increase especially when RDF data source is applied to social network. Since SPARQL 1.1 standard was proposed in 2013, there are still huge vacancies remained to comprehensively support those update operations. For the query performance, Map reduce paradigm [20] [30] can be intuitively adopted to parallelize the query tasks. However the iteratively synchronous communication protocol as well as the disk-based processing hinders the efficient data query. TriAD [12] and Trinity.RDF [29] thus employ a communication protocol based on MPI standard, yet the supplementary machine adding or removing will give rise to the failed or evicted query job. To this end, the elastic adjustment of cluster size should be autonomously conducted to satisfy the data dynamicity and transient surge.

Additionally, reliability is another key concern for data storage and query system due to increasingly common failures which are now the norm rather than the exception caused by the uncertain of system condition, enlarged system scale, and plethora of software or hardware faults that will activate [31]. Especially for the in-memory data store, severe faults such as main memory thrashing, out of memory (OOM) etc. will degrade the holistic system QoS [10]. How to guarantee

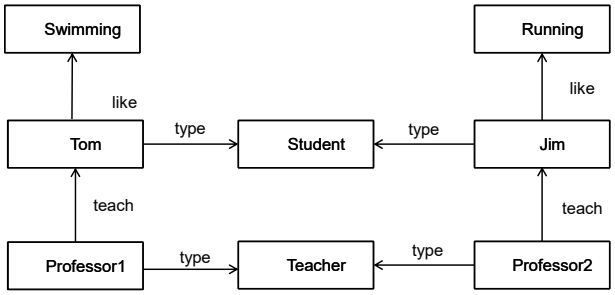


Fig. 1: An example of RDF graph

the user data and query tasks free from such system failures remains an intractable challenge.

In this paper, we describe ScalaRDF – a distributed and scalable RDF triple store based on in-memory key-value store. We present a novel protocol that extends the consistent hashing protocol [15] to achieve efficient data placement and elastic resource scale out/in. This approach allows for low-cost data store in event of RDF data update and dynamic cluster resource joining-in or departing. In our system, merely local data re-distribution will manifest, avoiding the holistic data oscillation. We also leverage a light-weight log-based mechanism to perform data operations in a batch manner. Through data redundancy mechanism, the in-memory data will be backed-up on the disk of adjacent nodes, minimizing the negative impacts under failure conditions. The system is implemented based on Redis [28] and Sesame [8] for data query engine. The experiments demonstrate that query time and update time are reduced by 87% and 90 % respectively compared to other approaches. For an 18G source dataset, the data redistribution takes at most 60 seconds when system scales out, and at most 100 seconds is needed for recovery when nodes experience crash-stop failures. In particular, the major contributions of our work can be summarized as follow:

- We firstly design a data placement and store protocol by extending the consistent hashing protocol. The data can be managed in a distributed in-memory cluster which can autonomously scale in-and-out with minimized data re-partition overhead.
- Our system can efficiently deal with updating operations of RDF data source and support triple record *insert* and *delete*, allowing for a large number of online RDF triple updates. The light-weight log-based mechanism helps the reduction of time-consuming delete operations.
- We realize a rapid data and service recovery mechanism through replication. We store in-memory data replicas in the next hop node, achieving transparent and rapid failover for running data queries.

The rest of this paper is organized as follows: Section II introduces an example and the system overview; Section III outlines our design philosophies and the main implementation details; Section IV presents the system evaluation; Section V discusses the related work; Section VI and VII discusses the current limitations and the conclusions of this work.

II. Motivated Example and Overview

A. Motivated Example

To make readers better understand our work, herein we give a simple example of how RDF system works. Figure 1 shows a RDF graph, describing the entities in a university (e.g., professors, students, courses and hobbies etc.) and the relationships among these entities (eg. teach, like). RDF store manages all the entities in memory or disk. When the user wants to find a student who likes swimming, and is supervised by Professor1, the query can be converted into a SPARQL query:

```
select ?x where {
?x type Student .
?x like Swimming .
Professor1 teach ?x .}
```

In particular, *<?x type Student>*, *<?x likes Swimming>* and *<Professor1 teach ?x>* are triple patterns. Each triple pattern will be matched in the RDF graph. The query result of the first triple pattern includes Tom and Jim. The second includes Tom and the third includes Tom. As these three triple patterns share the same variable x, all intermediate results will be joined together and generate the final result Tom. Consequently, the user will get the answer Tom.

B. Architecture Overview

Figure 2 illustrates the system architecture overview. The architecture of ScalaRDF follows a typical master-slave model. The master is responsible for parsing RDF files and distributing triples to different slaves. It also parses SPARQL query and generates the query plan subsequently. Correspondingly, the slaves take charge of storing triples and executing query plans. All the processes of execution and storage are monitored by the monitor component. When a node enters or leaves the cluster, Monitor will notify the master and the master will reschedule and drive the data repartition and placement. In particular, ScalaRDF mainly consists of the following components:

RDF Parser – It takes the responsibility of parsing RDF files (represented in TTL/N3 format), which reads files and extracts triples.

Dictionary – The triples generated by RDF Parser are originally in string format and each part of the triple (eg. subject, predicate, or object) is replaced by an ID to reduce the space occupation. The mapping of *<ID, String>* is stored by the dictionary component.

Partitioner – Triples generated by Dictionary will be converted into 6 indices (i.e., *SP_O, PO_S, SO_P, P_SO, O_SP, S_PO* key-value pairs). These 6 indices will be distributed to slaves to enable the consistent hashing protocol to perform on the occasion of data insertion.

SPARQL Parser & Query Plan & Optimized Query – The parser is responsible for preprocessing incoming queries and the query will be converted into a query plan. The plan will be separated into parallel compute tasks and executed in the cluster.

Deletion Logger – Considering the triple delete is very time-consuming, we utilize a log-based approach to record

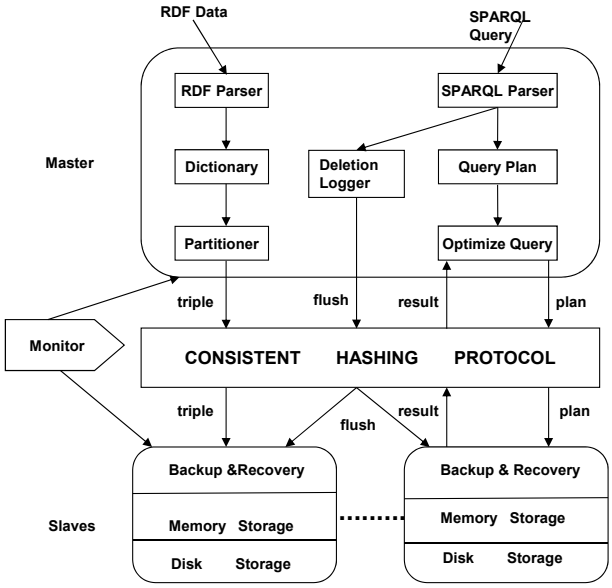


Fig. 2: System architecture

the operations (such as delete) in disk, staging the data and flushing to the backup file in a batch manner. The log is recorded in the master in case of data loss.

Backup & Recovery – Each slave stores indices in memory and backups them in disk. When a node gets failed, the backup data will be recovered and repopulated into the memory without interrupting the query service.

Memory & Disk Storage – Each slave stores the indices in local memory and backups the replicas in other slave's disk.

Monitor – Once a node joins-in and departs, it will notify corresponding nodes to deal with the events and trigger further recovery and adjustment.

III. Design and Implementation

A. Index and Data Placement

All RDF stores can be searched by using SPARQL queries that are composed of triple patterns. A triple pattern resembles a triple, while S, P and/or O can be variables or literals. To satisfy all query situations, it needs six indices, namely *SP_O, SO_P, OP_S, S_PO, P_SO O_SP* key-value pairs, according to [18]. As for *<?S, ?P, ?O>* pattern, which selects all triples, we store the source RDF file in master to support this triple pattern due to its infrequent usage. Table 1 shows the detail of indices. For example, as to the triple *<S1, P1, O1>*, for SP_O index, key = *hash ("1"+S1 +P1)*, value=O1; for S_PO index, key= *hash ("6"+S1)*, value=P1+O1. The performance of Six key-value pair indices will surpass that of Three Table Index [18] in lookup speed, as the time complexity is O(1).

After six key-value pair indices are generated, these indices will be distributed to slaves according to the consistent hashing protocol. For example, for a $2^{32}-1$ hash space, we compute the hash value of the node IP, which determines the position of the node in the hash ring. The key of the index decides the position of the triple. The index will be stored in the first node in clockwise, whose hash value is no less than its key. The

TABLE I: Six key-value pair indices

prefix	index type	key	value	triple pattern
1	SP_O	SP	O	<S, P, ?O>
2	PO_S	PO	S	<?S, P, O>
3	SO_P	SO	P	<S, ?P, O>
4	P_SO	P	SO	<?S, P, ?O>
5	O_SP	O	SP	<?S, ?P, O>
6	S_PO	S	PO	<S, ?P, ?O>

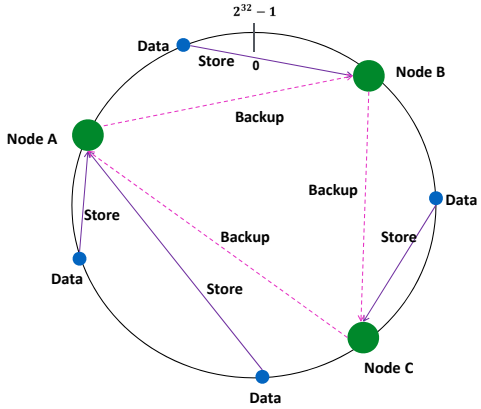


Fig. 3: RDF data storing and placement

index is stored in memory to improve the query performance. As discussed in [12], a memory storage cluster is sufficient to store the current biggest RDF dataset. After the dictionary component converts the string into ID, a source RDF file will be compressed when stored in memory. Redis is a key-value in-memory database and provides fast reading and writing capabilities, thus we use it as the memory storage.

In case of nodes getting failed, the triples are backed-up in disk. According to the consistent hashing protocol, when a node gets failed, the data stored in it should be stored in the next-hop node (next node in clockwise in hash ring). In this paper, we backup the in-memory data in the next-hop node's disk (we call this Next Hop Backup Rule). When this node gets failed, the next-hop node loads the data into memory to recover the system. This method will reduce the network communication and improve recovery performance. Figure 3 describes the storing organization.

B. Elastic Resizing of Compute Cluster

Users will also insert and update triples into the cluster using SPARQL query language continuously and the in-memory storage resource will be consumed. Besides, when the cluster becomes overloaded due to millions of query requests or workload surges [25], new nodes need to be automatically added into the cluster to timely supplement more computing resource. To minimize the negative impact onto the running tasks or queries, we design an optimized consistent hashing protocol to achieve dynamic and online new nodes adding without any downtime to services.

Each time when a node joins-in or departs the cluster, there are two steps to follow: one is to ensure the in-memory data consistency according to our consistent hashing protocol, and the other is to maintain the consistency among data replicas according to the Next Hop Backup Rule. All of them are to

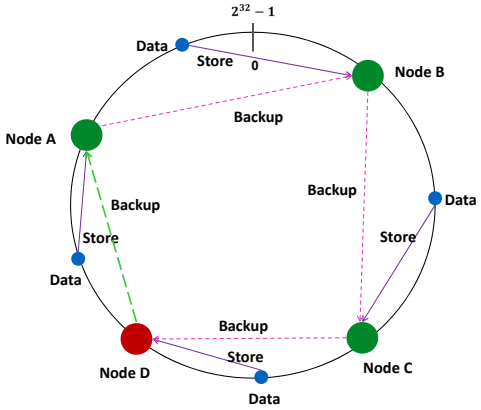


Fig. 4: System scale out

perform data re-distribution. Obviously, only the data stored within a specific range of nodes will be influenced, thereby minimizing the data re-distribution operational costs. Figure 4 demonstrates an example to show the detailed procedure:

1) *New node backups previous hop node's in-memory data:* For example, if Node D is added, the backed-up data of Node C has to be migrated from its previous Node A to Node D.
2) *Next hop node deletes the duplicated in-memory data:* After Node D is added, Data_D between Node C and Node D should be redistributed from its original repository Node A to Node D according to the decision made by the consistent hashing protocol. Node A should delete Data_D to avoid duplication (detailed in Algorithm 1). Subsequently, Node A checks the Deletion Log File and finally removes the corresponding triples out of its memory.
3) *New node loads data into memory:* According to the protocol, files between Node A and C are originally stored within the memory of Node A. After Node D is added, the data should be split and a fraction of the data will be re-distributed onto Node D. Similarly, the stale replicas on Node A should be removed to follow the consistency rule.

In practice, these steps can be executed in parallel. Once the third step finishes, the query service becomes available again and the system will recover after all steps completion. The Monitor component will monitor the cluster status in a real-time manner and notify the master with regard to the node adding event, and the master coordinates with corresponding slaves to recover the system.

C. Dynamic Updating Operations

Our system is implemented based on Sesame [8], a well-known framework for processing RDF data. The entire query processing workflow is outlined in Figure 5. The SPARQL parser accepts a SPARQL query as input and generates corresponding query model, which consists of triple patterns and join operations. The execution order of triple patterns will affect query performance significantly. The query optimizer uses cardinality generated in the step of data loading for query optimization and generates an optimized query model afterward. To accelerate the triple pattern matching, we execute

Algorithm 1 Delete duplicated data in memory

Define:
backward_key ← the hash key of the adding node's previous hop node in hash ring in clockwise
forward_key ← the hash key of the adding node

```
1:  keys = get all keys in memory
2:  for key:keys do
3:      if forward_key < backward_key then
4:          if backward_key < key then
5:              delete key //delete this key-value pair
6:          else if forward_key >= key then
7:              delete key //delete this key-value pair
8:          end if
9:      else
10:         if key <= forward_key && key > backward_key
    then
11:             delete key //delete this key-value pair
12:         end if
13:     end if
14: end for
```

the matching operation in parallel. Finally the query result will be returned to the user.

We extend SPARQL parser to support *insert* and *delete* operations. An example of SPARQL insert language is like:

```
INSERT DATA {student1 type student. student1
    like swimming.}
```

SPARQL parser decomposes it into two triples <*student1 type student*> and <*student1 like swimming*>, and transfers these triples to all slaves. In addition, slaves extract each triple, create indices for them, store the index belonging to itself according to the consistent hashing protocol, and backup the data belonging to the previous hop node in the hash ring. Before storing triples in memory, we will estimate if the triples is existed and append the non-repetitive triples at the end of the backup file.

Similarly, a SPARQL delete language is like:

```
DELETE DATA {student1 type student. student1
    like swimming.}
```

SPARQL parser decomposes it into two triples <*student1 type student*> and <*student1 like swimming*>, creates six indices for each of them, stores them directly into local log file (called the Deletion Log File), and then deletes them from the in-memory cluster. Deleting a triple is a very time-consuming operation because it has to traverse the backup file, find them and delete them. Despite the delete operations are low-frequent, a huge number of few triples are involved to store in backup file. To overcome the overhead of traversing backup file, we are inspired by the design of journaling file system. The master will store the indices locally. When the log file is filled or the system recovers from scaling in/out, the log file will be flushed to the backup files in slaves immediately. All the query requests during the update operation will be cached in a queue. Once the update operation finishes, the query requests will be reprocessed.

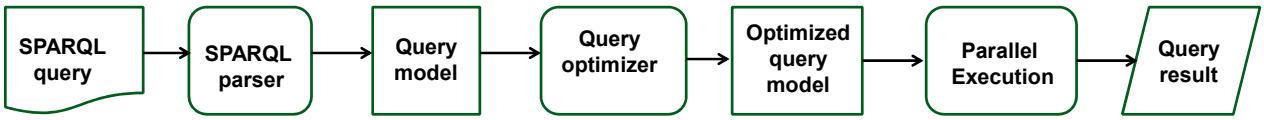


Fig. 5: SPARQL query processing workflow

Algorithm 2 Backup in-memory data

```
1: keys = get all keys in memory
2: for key:keys do
3:     for value:Redis.smember(key) do
4:         buffer.append(key);
5:     end for
6: end for
7: open backup file
8: write buffer to backup file
9: close backup file
```

D. System Data Recovery

The crash-stop of a server stemmed from software crash or late-timing failure (such as timing-out server with interrupted heartbeats) can be regarded as an equivalence of the cluster tailing-off. Another equivalent reason is the over-provisioned resource that can be removed for the energy-efficiency. In this paper, we will deal with these two cases in the same way: recovery the in-memory as well as the backup data. Figure 6 shows an example.

1) *In memory data recovery:* Before Node D crashed, the replica of File_D is stored in the disk of Node A. When Node D leaves, Node A loads File_D into memory. Node A then moves File_D to Node B, and Node B merges File_D and File_A together to keep the backup data consistent. Finally, Node A will check Deletion Log File and delete the corresponding triples out of its memory.
2) *Backup data recovery:* When Node D leaves, File_C stored in Node D will experience data loss. At this time, Node A will substitute Node D to backup the data in memory of Node C (detailed in Algorithm 2).

When writing this paper, we merely consider the single failure scenario in the proposed mechanism. The correlated and simultaneous failure tolerance and recovery techniques are currently in progress and we briefly discuss them in Section VI.

IV. Experiments and Evaluation

A. Experiment Setup

We conduct the experiments on a local cluster with 13 computing nodes that connected with 1Gb/s Ethernet. Each node is with two 1200 MHz processors, 252GB memory, 423GB hard disk and runs Debian 7.4, Apache Maven 3.2.1, Redis 3.0.7, Zookeeper 3.4.6, and Java 1.6.0_27. One node is taken as the central master while other nodes as slaves. LUBM [2] is used as the benchmark for our experimentation and it uses university as basic unit with roughly 23MB size. Seven queries (shown in Appendix) are designed to assess the query

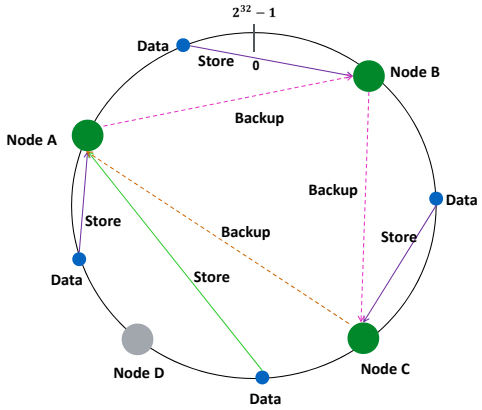
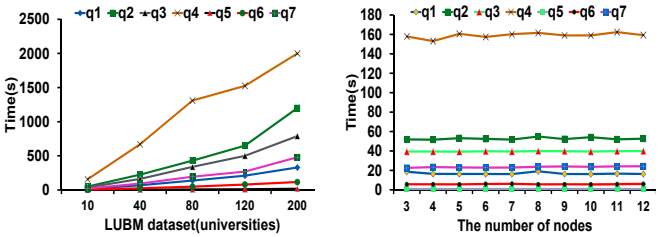


Fig. 6: Fault tolerance

performance [2] [12]. We use the following metrics to illustrate ScalaRDF outperforms others:

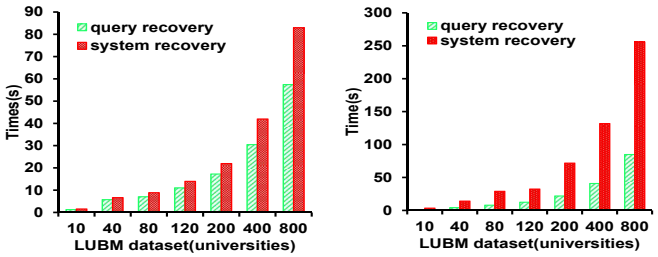
- **Query Performance –** It is measured by the time from inputting a SPARQL query to outputting the result.
- **Data Redistribution (Recovery) Performance –** Some data will be re-distributed after cluster re-sizes or system failovers. We therefore define two metrics: *query recovery time* and *system recovery time*. Query recovery time is measured by the time from a node being added or getting failed to the query service becomes completely re-available. System recovery time is measured by the query recovery time plus the additional time consumption to achieve eventual data consistency.
- **Update Performance –** Insert performance is measured by the time of storing triples into memory and appending triples into backup files. Correspondingly the delete performance is measured by the time of removing triples out of memory and generating logs into Deletion Log File.

We mainly compare the query performance of ScalaRDF with Rainbow [11]. Rainbow is a distributed and hierarchical RDF triple store with dynamic scalability and it uses HBase Cluster as the persistent storage, which creates Three Table Index [18] to support any type of triple patterns. Additionally, Rainbow takes Redis cluster as the second storage, which creates SP_O and PO_S indices to support frequently-issued triple patterns. When a Redis node gets failed, it automatically switches to HBase for querying. As Rainbow is not open source, we implement it and make a comparison in terms of the query performance. We do not compare the update performance due to the nonsupport by Rainbow. Moreover, we compare the operation performance of ScalaRDF with a typical centralized RDF system gStore [32].



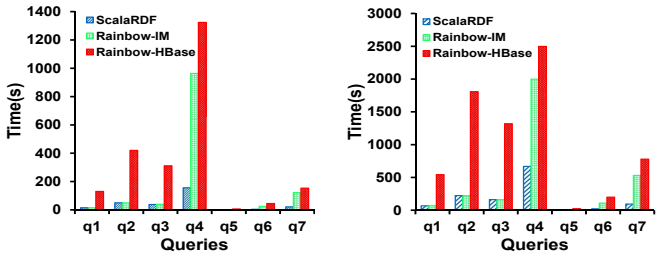
(a) Varying dataset (b) Varying nodes

Fig. 7: ScalaRDF query performance



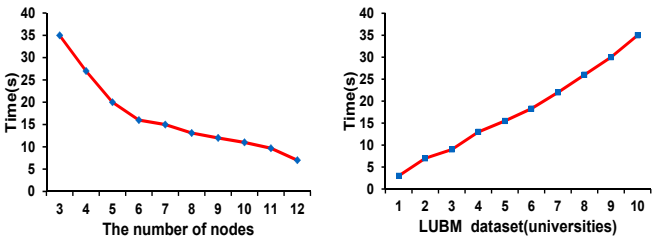
(a) Scale out (b) Fault tolerance

Fig. 9: Recovery performance



(a) LUBM-10 result (b) LUBM-40 result

Fig. 8: Query performance comparison



(a) Varying nodes with LUBM-10 (b) Varying dataset

Fig. 10: Insert performance

B. Query Performance

Figure 7(a) shows that the query time of ScalaRDF increases with the increment of dataset. This is because the query result size increases linearly when LUBM dataset grows. It is also observable from Figure 7(b) that ScalaRDF achieves stable performance regardless of the fluctuation of machine number due to the adopted centralized architecture of the query engine. To illustrate the performance improvement, we use LUBM-10 and LUBM-40 as exemplified test cases and make comparisons with other approaches: a) *Rainbow-HBase* which takes HBase as the storage layer without Redis cluster, and b) *Rainbow-IM* which leverages HBase as its persistent storage layer and Redis cluster as the cache layer.

As depicted in Figure 8, the performance of ScalaRDF and Rainbow-IM is approximately the same when performing q1, q2, q3, q5 queries, as those queries merely contain *SP_O, PO_S* triple patterns and the corresponding indices are all stored in Redis, indicating no significant difference. Nevertheless, ScalaRDF significantly reduces the query time by roughly 83% compared to Rainbow-IM in q4, q6, q7 queries. This phenomenon can be attributed to *SP_O, PO_S* triple patterns within the queries that have to read from HBase and Redis. Compared with Rainbow-HBase, ScalaRDF can constantly decrease the query time by approximately 87% in all cases due to the improvement of memory read speed.

C. Recovery Performance

As illustrated in Figure 9, the recovery time grows linearly with the dataset increment. In LUBM-800 benchmark (roughly 18GB size), the average time of query recovery is approximately 60 seconds and the corresponding time used to repopulate system states and reach eventual data consistency is at most 90 seconds in the event of new node adding. In contrast, to tackle server failover or cluster size shrinking, the query recovery time and system recovery time on average are roughly 100 seconds and 250 seconds respectively. Such data re-distribution can be still completed at minutes-level, thereby being soundly accepted for most RDF applications.

D. Operation Performance

Figure 10(a) illustrates the average time of insert decreases linearly with the increment of nodes. The insert operation is pre-stored in the disk of every slave. Each slave only needs to store triples assigned to allocate on itself according to the consistent hashing protocol, and needs to snapshot the data pertaining to the previous hop node in hash ring. When the total data to insert is fixed, the data partition that the node should store will decrease with the increment of node number, resulting in the decreased insert time. We also evaluate the impact of varying dataset on the insert performance. Similar to the query and recover effects, the increased data size will give rise to an augmented makespan (See Figure 10(b)). Due to the support of only insert operations in gStore, we only evaluate the insert performance with ScalaRDF. As demonstrated in Figure 11, the insert time of our approach can be significantly reduced by 90% compared to gStore. The reason for this phenomenon is that gStore has to traverse the whole data to find whether the triple exists. Instead, Redis implements $O(1)$ time complexity to determine the existence of a specific triple.

Figure 12 shows the delete performance result. The time will also grow linearly with the increment of dataset. In reality, the process of triples storing and file logging is controlled by the central master, therefore the varying nodes will have little impact on the result. According to our system design, the delete operations can be delay-executed according to the logs that record the triples metadata is about to delete, rather than directly traversing the backup file and subsequently deleting . In this manner, we can observe that the delete operation of a LUBM-10 benchmark (around 226M) merely takes 100 seconds. The minutes-level operation cost would be acceptable in most scenarios.

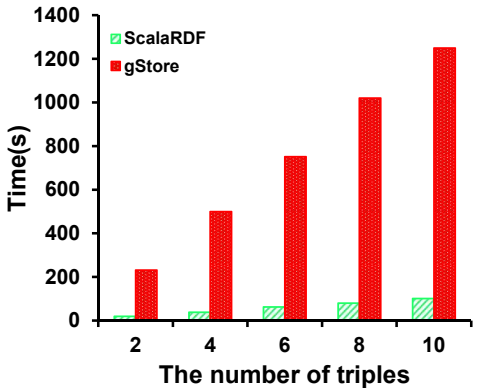


Fig. 11: Insert performance when compares with gStore

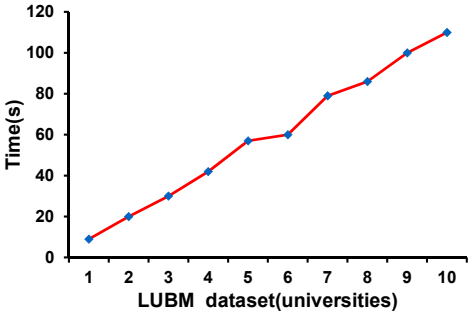


Fig. 12: Delete performance

V. Related Work

Tremendous efforts have been devoted to building high performance RDF management systems. We briefly introduce some of the most related work.

Relational and Native Graph Approaches – Most of the existing RDF stores, both centralized and distributed, use a relational model to manage RDF data. For example, SW-store [4], vertically partitions the RDF triples into multiple property tables, while Hexastore [22] and Dream [13] employ index-based solutions by storing triples directly in B+ trees. On the other hand, a number of approaches are proposed to store RDF triples in native graph format. These approaches employ adjacent lists as a basic structure for storing and processing RDF data. By using sophisticated indexes, centralized RDF graph engines such as gStore [32], BitMat [5] and TripleBit [27] prune many triples before invoking relational joins to finally generate the results of a SPARQL query. Trinity.RDF [29] is a typical distributed RDF graph engine by using graph exploration approach to fast evaluate SPARQL queries. Although Trinity.RDF supports many general-purposed graph-based operations (such as random walk, reachability, community discovery), it does not allow for the integration of parallel join techniques for non-selective queries. In this paper, ScalaRDF employs a relational model storage.

RDF systems based on native graph approaches usually use graph partitioning techniques to optimize the query evaluation. Apart from centralized RDF system such as gStore [32], BitMat [5] and TripleBit [27], [14] also follows a graph partitioning approach over a cluster, where triples are assigned to different nodes using the METIS [16] graph partitioner. Graph partitioning allows triples that are close to each other in the RDF data graph to be stored at the same machine, thus overcoming the randomness issued by the purely hashing-based partitioning schemes used in system such as SHARD [19]. However, the bottleneck of the graph partition store lies in the non-negligible repartition overhead when new batches of data or machines added.

Centralized and Distributed RDF System – Most centralized systems [4] [17] [22] utilize relational models to manage RDF data, while other systems [5] [27] [32] maintain RDF data in native graph format. These centralized RDF systems use sophisticated indexes to improve the query performance. To overcome the limited capacity of computing and storing resource of a standalone machine, many distributed RDF systems have been proposed. Based on the MapReduce paradigm, distributed RDF systems such as H-RDF-3X [14] and SHARD [19] horizontally partition an RDF dataset over a cluster of computing nodes and employ Hadoop as a communication layer for queries. However, MapReduce frameworks are known to incur a non-negligible overhead due to their iterative, synchronous communication protocols and disk-based computing schema. To overcome these shortcomings, Trinity.RDF takes MPI [21] as the communication protocol. It stores RDF triples in native graph forms and replaces joins with graph explorations. TriAD presents the asynchronous inter-node communication using MPI and parallel/distributed join execution over six in-memory indices. SparkRDF [9] uses Spark to implement SPARQL query. All the intermediate results are modeled as Resilient Discreted Subgraph to support fast iterative join operations. Although these distributed RDF system achieve excellent query performance, they do not have specific mechanism to support system scaling in/out and effective fault tolerance. Rainbow [11] is the first research work to solve this problem. It uses HBase Cluster as the persistent storage and takes Redis cluster as the second storage. It depends on HBase to achieve persistent storage's fault tolerance and scaling in/out, and employs consistent hashing protocol to ensure Redis cluster's dynamically scalability.

VI. Discussion

The work in this paper primarily focuses on provisioning a reliable and elastic RDF storage system, but there are still some limitations that need further improvement:

Centralized query engine – Although in-memory storage schema is able to speedup the query compared to the disk-based storing schema, the query can be further improved from current centralized design. In reality, all query requests are handled through the centralized master while the slaves in the cluster only acts as back-end RDF storages. This design will increase the possibility of single-point failure and become the bottleneck of request handling. the implementation of fully distributed engine is in progress.

Crash-stop failure model – We extend the consistent hashing protocol in this paper to leverage the data replication to handle single-point failure of nodes. The data on the failed server can be repopulated from the snapshot that stored in adjacent node in the event of server crash-stop failure. However, current ScalaRDF cannot handle simultaneous and correlated failure scenario where a combinations of failures will manifest. To this end, we plan to transplant our previous

works [24] [26] into this RDF store to handle more general and sophisticated failures including both crash-stop failure and late-timing failure etc. We are also developing a probability model to improve the placement of replicas according to the failure probability of each node in real-system.

Imbalanced data distribution – As an initial system implementation, we merely optimized the consistent hashing protocol without applying virtual nodes to reduce costs. All nodes in the hash ring are physical machines, resulting in the imbalanced data distribution. Ketama hash algorithm [1] can be referred as a standard implementation of consistent hashing protocol to make triple store balanced via employing virtual node. We plan to extend our work on the basis of the algorithm in the future.

VII. Conclusions

In this paper, we introduce ScalaRDF, a distributed, elastic and scalable in-memory RDF triple store. A novel consistent hashing protocol is proposed to make RDF data placement decision, take responsibility for data operations for editing web contents and allow for an autonomous data re-distribution in the event of cluster node joining or departing. In addition, the data store is able to rapidly recover based on the replicated data that backed-up in the next hash hop to realize a user transparent failover. The experiments demonstrate that the query time and update time are reduced by 87% and 90 % respectively compared to other approaches. For an 18G source dataset, the data redistribution takes at most 60 seconds when system scales out, and at most 100 seconds is needed for recovery when nodes experience crash-stop failures.

Acknowledgments

This work is supported by National Key Research and Development Program (2016YFB1000103), National 863 Program (2015AA01A202), and China Natural Science Foundation (NSFC) (No. 61421003). This work is also supported by Beijing S&T Committee as part of Beijing Brain Inspired Computing Program in BCBD innovation center.

References

[1] Memcached. https://memcached.org/.

[2] SWAT Projects the Lehigh University Benchmark(LUBM). http://swat.cse.lehigh.edu/projects/lubm/.

[3] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the VLDB Endowment*, 2007.

[4] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 2009.

[5] M. Atre, J. Srinivasan, and J. Hendler. Bitmat: A main-memory bit matrix of rdf triples for conjunctive triple pattern queries. In *Proceedings of the 2007 International Conference on Posters and Demonstrations-Volume 401*. CEUR-WS. org, 2008.

[6] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[7] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of ACM SIGMOD*, 2013.

[8] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International semantic web conference*, pages 54–68. Springer, 2002.

[9] X. Chen, H. Chen, N. Zhang, and S. Zhang. Sparkrdf: elastic discreted rdf graph processing engine with distributed memory. In *2015 IEEE/WIC/ACM WI-IAT*, 2015.

[10] P. Garraghan, X. Ouyang, R. Yang, D. Mckee, and J. Xu. Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters. *IEEE Transactions on Services Computing*, 2016.

[11] R. Gu, W. Hu, and Y. Huang. Rainbow: A distributed and hierarchical rdf triple store with dynamic scalability. In *Proceedings of IEEE BigData*, 2014.

[12] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of ACM SIGMOD*, 2014.

[13] M. Hammoud, D. A. Rabbou, R. Nouri, S.-M.-R. Beheshti, and S. Sakr. Dream: distributed rdf engine with adaptive query planner and minimal communication. *Proceedings of the VLDB Endowment*, 2015.

[14] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 2011.

[15] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11):1203–1213, 1999.

[16] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 1998.

[17] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, pages 91–113, 2010.

[18] R. Punnoose, A. Crainiceanu, and D. Rapp. Rya: a scalable rdf triple store for the clouds. In *Proceedings of the ACM 1st International Workshop on Cloud Intelligence*. ACM, 2012.

[19] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In *ACM Programming Support Innovations for Emerging Distributed Applications*, 2010.

[20] K. Rohloff and R. E. Schantz. Clause-iteration with mapreduce to scalably query datagraphs in the shard graph-store. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*. ACM, 2011.

[21] D. W. Walker and J. J. Dongarra. Mpi: a standard message passing interface. *Supercomputer*, 1996.

[22] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 2008.

[23] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: a probabilistic taxonomy for text understanding. In *ACM SIGMOD*, 2012.

[24] R. Yang, T. Wo, C. Hu, J. Xu, and M. Zhang. Dˆ 2ps: A dependable data provisioning service in multi-tenant cloud environment. In *IEEE HASE*, 2016.

[25] R. Yang and J. Xu. Computing at massive scale: Scalability and dependability challenges. In *IEEE SOSE*, 2016.

[26] R. Yang, Y. Zhang, P. Garraghan, Y. Feng, J. Ouyang, J. Xu, Z. Zhang, and C. Li. Reliable computing service in massive-scale systems through rapid low-cost failover. *IEEE Transactions on Services Computing*, 2016.

[27] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: a fast and compact system for large scale rdf data. *Proceedings of the VLDB Endowment*, 2013.

[28] J. Zawodny. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine*, 79, 2009.

[29] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *Proceedings of the VLDB Endowment*, 2013.

[30] X. Zhang, L. Chen, Y. Tong, and M. Wang. Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud. In *Proceedings of IEEE ICDE*, 2013.

[31] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *Proceedings of the VLDB Endowment*, 7(13):1393–1404, 2014.

[32] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: answering sparql queries via subgraph matching. *Proceedings of the VLDB Endowment*, 2011.

APPENDIX

LUBM queries

```
PREFIX rdf: http://www.w3.org/1999/02/22-rdf-
    syntax-ns\#
PREFIX ub: http://www.lehigh.edu/~zhp2
    /2004/0401/univ-bench.owl\#

q1: select ?x where{ ?x rdf:type ub:
    GraduateStudent . ?x ub:takesCourse http://
    www.Department0.University0.edu/
    GraduateCourse0}

q2: select ?x where{ ?x rdf:type ub:
    Publication . ?x ub:publicationAuthor http:
    //www.Department0.University0.edu/
    AssistantProfessor0}

q3: select ?x where{ ?x rdf:type ub:
    UndergraduateStudent}

q4: select ?x ?y ?z where{ ?x ub:memberOf ?z .
     ?z ub:subOrganizationOf ?y . ?x ub:
    underGraduateDegreeFrom ?y }

q5: select ?x where{ ?x rdf:type ub:
    FullProfessor .}

q6: select ?x ?y where{ ?y rdf:type  ub:
    Department . ?x ub:worksFor ?y . ?x rdf:
    type ub:FullProfessor .}

q7: select ?x ?y ?z where{ ?y ub:teachOf ?z .
    ?y rdf:type ub:FullProfessor . ?z rdf:type
     ub:Course . ?x ub:advisor ?y . ?x rdf:type
     ub:UndergraduateStudent . ?x ub:
    takesCourse ?z .}
```