論文 / 著書情報
Article / Book Information

Title	JARS: Join-Aware Distributed RDF Storage
Authors	Anjali Rajith, Shoji Nishimura, Haruo Yokota
Citation	Proceedings of IDEAS 2016, , , pp. 264-271
Pub. date	2016, 7
Note	(c) ACM, 2016. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in PUBLICATION, pp. 264-271, 2016, 7, http://dx.doi.org/10.1145/2938503.2938548.

JARS: Join-Aware Distributed RDF Storage

Anjali Rajith*
Tokyo Institute of Technology
Tokyo, Japan
ranjali@de.cs.titech.ac.jp

Shoji Nishimura
Tokyo Institute of Technology
Tokyo, Japan
nishimura@de.cs.titech.ac.jp

Haruo Yokota
Tokyo Institute of Technology
Tokyo, Japan
yokota@cs.titech.ac.jp

ABSTRACT

The enormous increase of data in RDF format calls for efficient storage and retrieval approaches. Being a highly connected data, RDF generates massive amounts of intermediate results during query processing. Many of the current RDF storage approaches involve large amounts of inter-node data movement even for simple selective query patterns. We propose JARS, a join-aware distributed RDF storage system with a dual-hash partitioning strategy coupled with two layered distributed clustered indexing and a rule-based query-execution approach. JARS eliminates the inter-node communication for star patterns and mitigates the communication cost for chain pattern SPARQL queries. Our experiments indicate that JARS achieves significant performance enhancement over the state-of-the-art RDF storage systems.

Categories and Subject Descriptors

A.m [**General**]: Database Management, Semantic Web

Keywords

RDF, SPARQL, Clustered Index, Dual-hash Distribution

1. INTRODUCTION

The Resource Description Framework (RDF) was originally designed as a metadata data model, by providing statements representing information in the web resources in a format that comprises a subject, a predicate and an object. The ⟨subject(s) ->predicate(p) ->object(o)⟩ relationship is called as triple, where the subject and object denote resources and the predicate expresses the relationship between subject and object. Spurred by the Linking Open Data Project [2], which links data using semantic web technologies, an ever-increasing volume of data are published in RDF format.

There is a flurry of research efforts aiming to build efficient high performance RDF data management systems. Various approaches adopted different storage strategies such as *in-memory storage*: RDF data is stored in main memory, *native storage*: provide dedicated persistent databases such as Virtuoso [3], and *relational storage*: relational tables are used to store RDF data such as Jena [8]. The in-memory stores are capable of storing only limited data. The native graphical approaches assume data to be static and require high preprocessing cost for partitioning.

We adopt the relational approach by taking into account of various advantages such as efficient storage and querying, transaction support, security, locking and compression which are not available in native stores. Within the relational approach, there are various physical organization techniques such as *triple-table*, *horizontal partitioning* and *vertical partitioning* approach. *Triple-table* stores the entire RDF data in a single 3-column table, where each row is a RDF statement. Though it performs well, the large size potentially requires large number of self-joins. The *horizontal* representation conceptually stores all the RDF data in a single table and the table dedicates a column for each predicate value in the RDF graph. This wide-table approach supports multi-valued attributes, but results in large number of empty cells due to sparse attributes. The *vertical partitioning* approach rewrites the triple table into n two-column tables where n is the number of unique properties in the data and, this approach is used in works such as Abadi et al; [1] . It is very easy to implement and performs well for queries that specify the predicate values, but else not a good approach.

In the past years, many researches focused on centralized RDF data management techniques such as Jena, RDF-3X [11] and Hexastore [18]. But, the rapid growth of data led to the requirement for scalable distributed storage systems. So, we attempt to implement a scalable distributed triple-table storage approach by overcoming the disadvantages of the triple-table through appropriate clustered indexes and data partitioning strategy. We distribute data among the cluster through a novel variant of hash partitioning technique, dual-hash partitioning coupled with distributed clustered indexing.

There have been some clustered RDF database systems such as Virtuoso [3], and SHARD [14], that adopted hash partitioning approach, but they have proven to work well only for star queries and simple index-lookup queries. For more complex queries, the results were far from optimal.

We introduce JARS, a **J**oin-**A**ware **R**DF **S**torage approach with two layered clustered indexes, with dual-hash distribu-

*The author presently works for Center for Technology Innovation, R&D Group, Hitachi Ltd., Kanagawa, Japan

IDEAS '16, July 11-13, 2016, Montreal, QC, Canada

 ISBN 978-1-4503-4118-9/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2938503.2938548

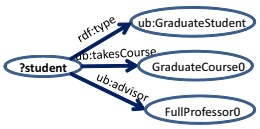

Figure 4: SPARQL star pattern

Figure 5: SPARQL chain pattern

tion of triples based on the subject and the object, coupled with alternative combination indexes for the subject, predicate and object. We particularly attempt to address the inter-node communication cost through efficient co-location of join-able triples, which greatly improves the query execution performance. As a result, star- and simple-chain SPARQL queries can be executed locally, whereas composite queries can be decomposed and executed in parallel. We avoid predicate-based hash partitioning, since most SPARQL joins involve *s-s, s-o, o-o* joins, where predicate joins are negligible. Besides, predicate-based hash distribution creates big skew due to the presence of many triples with the predicate *rdf:type*, thereby hampers optimized load balancing. Our approach helps to meaningfully partition the RDF data, minimize the communication cost to a large extent in majority of the queries and, improves query-response time.

SPARQL.

Simple Protocol and RDF Query Language, SPARQL [16] is the standard language for querying RDF data. It comprises a set of RDF triple patterns ($< s, p, o >$) called as Basic Graph Pattern (BGP), where each position can be a variable. Answering SPARQL involves matching the variables with the real constant values from the RDF graph. The two common patterns in SPARQL queries are star pattern and chain pattern. Former is the most common pattern, in which many triple patterns share the same subject, as shown in Figure 4, whereas later comprises a sequence of triple patterns in which the object of the previous pattern is the subject of the next pattern, as shown in Figure 5.

1.1 Contributions

We summarize the novelty and advantages of our work as follows.

i) We introduce a novel join-aware distributed RDF storage system, JARS with a dual-hash partitioning strategy, which replicates RDF triples based on hash of subject and object and proves that by doing so, join-able triples are localized. As a result, the common star and simple chain SPARQL patterns can be executed without inter-node communication, whereas complex queries can be decomposed and executed in parallel.

ii) We provide an easy-to-implement, scalable RDF storage paradigm that does not require any data preprocessing and maintenance cost.

2. RELATED WORKS

Here, we give a brief overview of relevant related works by discussing some efficient centralized and distributed RDF storage systems.

Centralized systems.

RDF-3X and Hexastore are highly efficient centralized RDF storage systems which implement exhaustive, but space-efficient clustered triple indexes for better performance.

Distributed Systems.

Distributed systems characterize a data partitioning strategy for improving the query performance. SHARD is a distributed approach that stores RDF triples directly in HDFS as flat files and runs one Hadoop [5] job for each triple pattern in the SPARQL query. A heuristic-based system [7] stores RDF triples in HDFS by hashing on predicates and runs one Hadoop job for each join in the SPARQL query. However, predicate-based hashing can lead to skewed storage. Virtuoso Cluster, and ClusteredTDB [12] are few other distributed systems which involve hash-partitioning strategy, but different from JARS. However, many of these approaches suffer from high communication costs.

Another approach adopted by Huang et al; [6] partitions RDF graph into smaller sub graphs through a n-hop strategy, and are stored in separate nodes running RDF-3X. This method guarantees parallel SPARQL processing within n-hop limits, but incurs huge computational cost. The distributed semantic hash partitioning approach [9] extends the simple hash partitioning method through direction-based triple groups and triple replications. However, this approach also requires significant preprocessing time and maintenance cost on data updates. Though JARS is similar to these approaches with respect to the data partitioning strategy, JARS doesn't incur the preprocessing cost. In JARS, adding / deleting data is very simple, hassle free and requires zero maintenance cost. Also, many of these systems handle data partitioning using graph partitioners such as METIS, which crashes for large data sets and therefore, limits the data size. The approach adopted by Wang et al; [17] combines the advantages of RDF-3X and Map/Reduce framework and is similar to JARS, with a query processing strategy to reduce the amount of intermediate results for minimized query latencies. But, it requires significant data preprocessing and additional data structures before performing data distribution to the respective nodes.

Problems of Relational RDF Storage Methods.

In the relational approach, especially in the triple-table method, answering a SPARQL query requires many self-joins. This leads to a performance bottleneck caused by the huge amount of data transfer among computing nodes. From the careful analysis of the SPARQL query patterns, composed of star and chain, we observed the large involvement of *s-s*, *s-o* and *o-o* joins. So, we devised a join-aware approach which co-locates triples with the same subject/object in the same server and avoids joins across nodes. Our join-aware approach is discussed in detail in section 3.2.

3. STORAGE AND ARCHITECTURE

We actualize a distributed storage architecture for JARS, as shown in Figure 6. JARS accepts triples in any RDF format and the data in the $< s, p, o >$ format is hashed twice based on the subject and object. The MD5 message-digest algorithm [10], a cryptographic hash function with a 128-bit hash value, is used to obtain the hash value of the subject and the object. The remainder of the division of

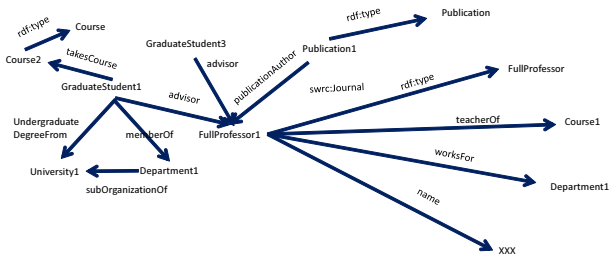

Figure 1: Sample RDF graph from LUBM Benchmark

Id	Subject	Predicate	Object
318	GraduateStudent1	rdf:type	GraduateStudent
319	GraduateStudent1	takesCourse	Course1
320	Publication1	rdf:type	Publication
321	FullProfessor1	rdf:type	FullProfessor
322	FullProfessor1	worksFor	Department1

Figure 2: Triple table - hashed by subject

Id	Subject	Predicate	Object
320	Publication1	publicationAuthor	FullProfessor1
549	Publication1	rdf:type	Publication
560	GraduateStudent3	advisor	FullProfessor1
569	GraduateStudent4	advisor	FullProfessor1
581	GraduateStudent5	advisor	FullProfessor1

Figure 3: Triple table - hashed by object

the resultant value by number of worker nodes would be the target location node of the respective triple.

The triples are therefore stored twice, with all triples that have the same subject/object, targeted to the same server. The database in each server contains two tables, one for holding all triples with the same hash value for the subject and the other for holding all triples with the same hash value for the object, as shown in Figures 2 and 3 with respect to the RDF Graph in Figure 1. The mechanism is demonstrated in Figure 7. Here, the subject of the triple is `Article123` and the hash value is 0. This triple is therefore directed to the subject table in Server0 based on the hash value of the subject. Similarly, the object is `PaulErdoes` and the hash value is 2 and is sent to the object table in Server2. The same triple is therefore routed to Server1 as well as Server2 based on the hash value of the subject and object. We use PostgreSQL database for our storage and concurrent batch insertion of data is performed. With the cost of storage reducing day by day, we can ignore the space cost involved in the replication.

3.1 Clustered Indexes

To improve query performance, the vertical represented tables are stored in multiple sorted orders based on permutations of the RDF elements $< s, p, o >$. Clustered BTrees help to retrieve results quickly due to availability of triples in multiple sort patterns and facilitate fast merge joins. However, they require much storage space and care should be taken to obtain a successful trade-off between the storage and the query efficiency.

However, given that the majority of the SPARQL joins involve *s-s*, *s-o* and *o-o* joins, this skew in the self-joins can be utilized to reduce the number of indexes.

The index patterns are carefully selected after analyzing join patterns in real life, in addition to the benchmark queries in the SP2 SPARQL Benchmark [15] and the Lehigh University Benchmark (LUBM) [4]. In JARS, the favored clustered indexes are selected based on the cardinality of the triple variables, **constant variables** with a single value, **join variables** with values sharing the same hash key, and

```
SELECT DISTINCT ?person ?name
WHERE { ?article rdf:type bench:Article.
        ?article dc:creator ?person.
        ?inproc rdf:type bench:Inproceedings.
        ?inproc dc:creator ?person.
        ?person foaf:name ?name }
```

Join Variable | Constant Variable | Free Variable

Figure 8: SPARQL variables

free variables which are literals. For example, for the SPARQL query shown in Figure 8, the join variables are `?article`, `?inproc` and `?person`, the constant variables are `rdf:type`, `bench:Article`, `bench:Inproceedings` and `foaf:name`, the free variable is `?name` and the projection variables are `?person` and `?name`.

Analysis of join pattern.

The triples in the query patterns can be either one-variable triples such as $\langle ?s,\ p,\ o \rangle$, $\langle s,\ ?p,\ o \rangle$ and $\langle s,\ p,\ ?o \rangle$, or two-variable triples such as $\langle ?s,\ ?p,\ o \rangle$, $\langle ?s,\ p,\ ?o \rangle$, and $\langle s,\ ?p,\ ?o \rangle$, or three-variable triples such as $\langle ?s,\ ?p,\ ?o \rangle$. The join patterns are analyzed and based on the position of hash variable, appropriate indexes are selected. Considering only the typical join patterns, *s-s*, *s-o* and *o-o* joins, we figured out three varieties of indexes: *Constant-Constant-Join (CCJ) index:* for patterns with two constants and one join variable, *Constant-Join-Free (CJF) index:* for patterns with one constant, one join and one free variable, and *Join-Free-Free (JFF) index:* for patterns with one join and two free variables. The hash variable is underlined to indicate the hash position, whether subject or object.

- Triple with one variable - CCJ-type index
  - For $\langle \underline{?s},\ p,\ o \rangle$ pattern, $po\underline{s}$ or $op\underline{s}$ can be used.
  - For $\langle s,\ \underline{?p},\ o \rangle$ pattern, $so\underline{p}$ or $os\underline{p}$ can be used.
  - For $\langle s,\ p,\ \underline{?o} \rangle$ pattern, $sp\underline{o}$ or $ps\underline{o}$ can be used.
- Triple with two variables - CJF-type index
  - For $\langle \underline{?s},\ p,\ ?o \rangle$ pattern, $ps\underline{o}$ can be used.

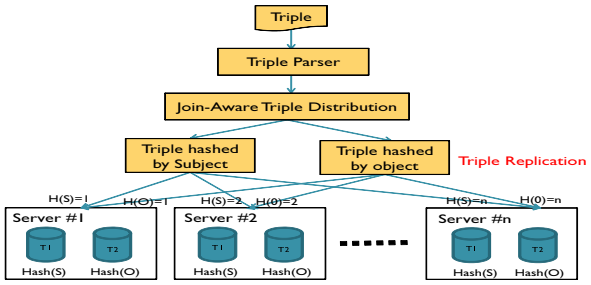

Figure 6: Storage Architecture

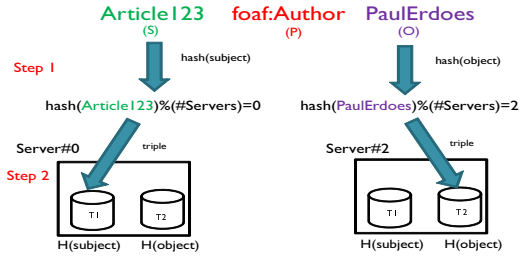

Figure 7: Example demonstration of triple distribution

  - For ⟨*?s*, *p*, *?o*⟩ pattern, *pos* can be used.
  - For ⟨*?s*, *?p*, *o*⟩ pattern, *osp* can be used.
  - For ⟨*?s*, *?p*, *o*⟩ pattern, *ops* can be used.
  - For ⟨*s*, *?p*, *?o*⟩ pattern, *sop* can be used.
  - For ⟨*s*, *?p*, *?o*⟩ pattern, *spo* can be used.

- Triple with three variables - JFF-type index
  - For ⟨*?s*, *?p*, *?o*⟩ pattern, *spo* or *sop* can be used.
  - For ⟨*?s*, *?p*, *?o*⟩ pattern, *osp* or *ops* can be used.
  - For ⟨*?s*, *?p*, *?o*⟩ pattern, *pso* or *pos* can be used.

Based on the triple-join-pattern analysis, and concentrating on the *s-s*, *s-o* and *o-o* joins, four clustered indexes are selected for each table: *pos*, *pso*, *osp* and *spo* are applied on the subject table and *pos*, *pso*, *sop* and *ops* on the object table. The indexes sop/osp, ops, spo, pso/ pos are neglected after considering only the *s-s*, *s-o* and *o-o* joins. This selection of indexes has two main advantages:

- We can perform an inexpensive merge join of the results from all servers after performing the join operations to obtain the end result.
- All SPARQL triple patterns that involve *s-s*, *s-o* and *o-o* patterns can be processed efficiently using a single index scan of the corresponding index.

As discussed earlier, a typical join pattern will involve a *s-s*, *s-o* and *o-o* pattern because they represent the majority of the patterns. Therefore, the triple patterns with predicate as the join variable are ignored.

3.2 Join-Aware Triple Distribution

Consider the example of a range-partitioned clustered index on a database table, where data is mapped to partitions based on the range of key values that are established for each partition. As illustrated in Figure 9, suppose we need to join the triples of the patterns, (s1, p1, ?v) and (?v, p2, o2), the candidate results may not be in the same server. (s1, p1, v1) and (v1, p2, o2) are stored in separate servers and would therefore require remote calls to perform the join operations and increase query latency.

The dual-hash partitioning coupled with the clustered indexing in the join-aware approach helps to eliminate remote calls to a great extent and thereby improves the query execution time. For example, consider the chain triple pattern in this SPARQL query with the join variable **?student**:

?x `rdf:type` **?student** (T1)
?student `ub:takesCourse ub:GraduateCourse0` (T2)

The candidate triples for both the triple patterns (denoted by **T**), T1 and T2 will be in the same server. Suppose the first triple pattern, T1 is stored in server1 based on the hash of object, the candidate results for the second triple pattern, T2 will also be stored in the same server, server1 based on the hash of the subject. This efficient retrieval of results by utilizing the clustered indexes, pso and the ops are illustrated in Figure 10. Moreover, the candidate result sets for the patterns (p1, s1, v1) and (o2, p2, v1) would be stored in the same server. Therefore, no remote calls are needed for the execution of this query.
Similarly, consider the BGP of a star pattern SPARQL query with join variable **?student**:

?student `rdf:type ub:GraduateStudent .` (T3)
?student `takes:Course ub:GraduateCourse0` (T4)
?student `ub:memberOf` ?z (T5)

Again, for this scenario, no intercommunication is required among the servers to obtain the result set. Therefore, this replication mechanism greatly mitigates the data transfer for *s-s*, *s-o* and *o-o* joins.

4. QUERY PROCESSING

Query processing is an extremely important aspect of the overall process, because SPARQL queries typically include multiple joins on the same or different variables. Therefore, we devised a rule-based join pattern analysis and execution, which basically performs concurrent bushy joins as shown in 11 as the initial step.

4.1 Query Processing Architecture

Before delving into the details of the query processing, the overall query processing architecture is discussed and illustrated in Figure 13. The SPARQL query provided by the user is processed in this order: Query Parsing, Query Conversion, Query Distribution, Result Redistribution and Aggregation and finally Output of Results.

4.2 SPARQL Query Parsing and SQL Conversion

In the first step, SPARQL Query Parsing, we use Jena Parser to confirm whether the query is syntactically accurate and convert it to an algebraic operator tree. After that, we convert it to SQL format.

4.3 Join Operations in SPARQL

The query processing in JARS comprises two phases. The first phase is the Join Pattern Scan Phase, in which the query planner scans the Basic Graph Pattern (BGP) to de-

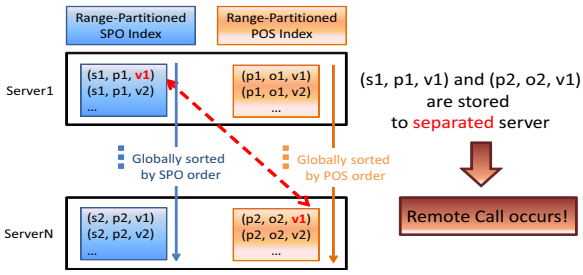

Figure 9: Range-partitioned Clustered Index

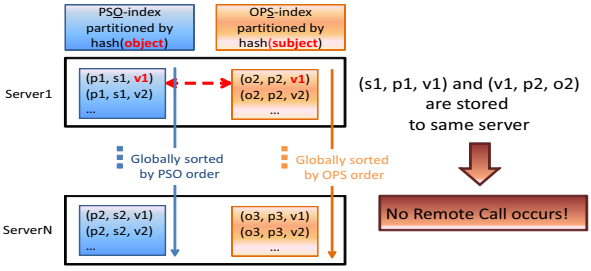

Figure 10: Clustered Index layout example in JARS

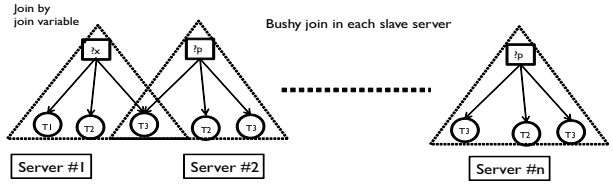

Figure 11: Bushy-Join performed in parallel at each server independently at first stage, and inter-node communication in second stage

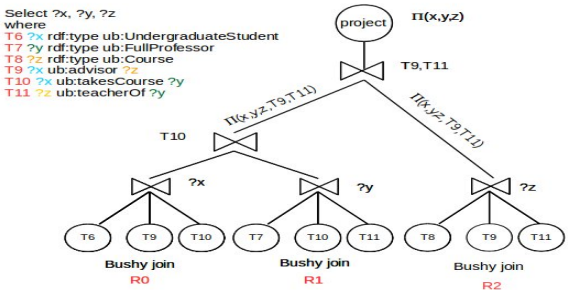

Figure 12: Query Processing Graph

termine the types of variables: join, constant, free and projection variables. Algorithm 1 is used to decide the join and query execution order for the query.

4.4 Query Processing Algorithm

We have devised a rule based query-processing algorithm, in which the primary step is to check whether the query can be computed without inter-node communication. So, we check the number of triple patterns in the BGP and the number of join variables. In this implementation, all queries with one join variable do not require any inter-node communication. In all other cases, the query planner estimates the query processing order based on the number of join variables, constants and the cardinality estimates, such that the amount of intermediate results is maintained minimal. Higher the number of intermediate results, higher the inter-node communication and query latency. Query-processing methods based on the number of triple patterns in the BGP (denoted as #BGP) and join variables are based on the following cases:

Case 1.

If the #BGP = 1, then the constant variable position in the SPARQL Algebra is checked. If the constant variable is in the subject position, the subject table present in the server is queried else vice-versa.

Case 2.

If the #BGP > 1 and #join variables = 1, the query planner checks for the join variable position in each triple pattern. If the join variable position is the subject position, the subject table in the nodes are queried (and object tables for object positions).

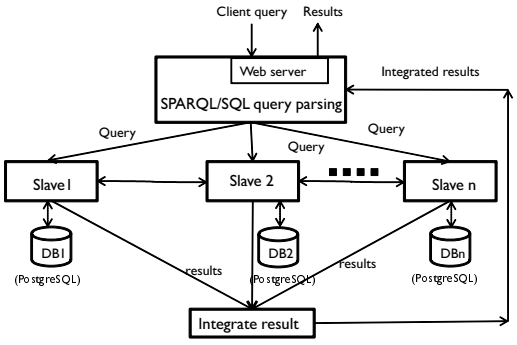

Figure 13: Query Processing Architecture

Case 3.

If #join variables ≥ 2, the query planner decomposes the query into sub-queries based on the join variable as shown in the query of Figure 12 and the number of join variables in each sub-query is estimated.

1) Check the number of join variables in each sub-query.

   (a) If any sub-query, SQ contains all the join variables, push it to the bottom of query-execution order. Process all other sub-queries concurrently and hash distribute the results based on the common join variable with SQ. Finally, merge join the results with remaining SQ that is not yet executed.

   (b) If all sub-queries contain all join variables, concurrently execute all the sub-queries and redistribute results based on combination of hash of common triple ids. In this case, *number of intermediate stages = number of sub-queries-1.*

(c) In all other cases, concurrently process all the sub-queries and redistribute results based on hash of common join variable.

Algorithm 1 Join Planner Algorithm

Input:SPARQL
Output: Query Processing plan

```
JV ← joinvariables
if JV = 1 then
    ConcurrentExecute()
end if
if JV ≥ 2 then
    for each JV do
        Decompose(tripleList)
    end for
    for each query in QueryList[] do
        getCount(JV)
        if getcount(QueryList[i].JV) == JV then
            Flag++
        end if
    end for
    if (Flag == JV) then
        QueryList[i].ConcurrentExecute()
        HashDistribute(result, hashkey(tripleid))
    end if
    if (Flag < JVandSubQuery[i].JV == JV) then
        ExecutionList.InsertLast(Subquery[i]
        ExecuteQuery(ExecutionList[i])
    else
        QueryList[i].execute()
        HashDistribute(result, hashkey(tripleid))
    end if
    BushyJoinExecute(QueryList[i])
    HashDistribute(result, hashkey(tripleid))
end if
```

Consider the LUBM query Q9, one of the complex queries with a triangular relationship among the sub-queries.

```
?x rdf:type ub:UndergraduateStudent .        (T6)
?z rdf:type ub:FullProfessor.                (T7)
?y rdf:type ub:Course.                       (T8)
?x ub:advisor. ?z                            (T9)
?x ub:takesCourse ?y                         (T10)
?z ub:teacherOf ?y                           (T11)
```

The above query would be split based on the join variables, `?x`, `?y` and `?z` to form three sub-queries with triple patterns, SQ1 (T6, T9, T10), SQ2 (T8, T10, T11), and SQ3 (T7, T9, T11). In the stage 1 of the query processing, each server processes the sub-queries as "bushy joins" in parallel, as shown in Figure 11. After stage 1, we obtain three result sets from each server, denoted R0, R1 and R2 as shown in Figure 12. At this stage, no communication is required among the servers.

A triangular relationship exists among the sub-queries; SQ1 and SQ2 with T10 in common, SQ1 and SQ3 with T9 in common and SQ2 and SQ3 with T11 in common. The join-variables **?x**, **?y** and **?z** are present in all the three sub-queries. We perform the concurrent execution of all sub-queries and the results are re-distributed based on the hash value of common triple ids, in such a way that the amount of intermediate results and the number of intermediate tables are reduced. The results of SQ1 and SQ2 are distributed based on the hash of common triple id, T10. The result set obtained after joining SQ1 and SQ2 and the results from SQ3 from each node are again redistributed based on the combination of hash of common triple ids (T9, T11) in SQ3. The final result set is obtained after performing merge join in each node. The entire process requires only 4 intermediate tables in each node.

SPARQL UNIONs are processed in such a way that the sub-queries within the UNION operator are treated individually and are concurrently processed. SPARQL OPTIONAL feature is implemented through the SQL-equivalent, left join.

To summarize the query processing, the query processing order is decided by the Join Planner Algorithm, presented in Algorithm 1. Each node executes the queries in parallel and the results are finally integrated to obtain the final result set.

4.5 Query Optimization

Frequency histograms and cardinality estimates of the intermediate result set play huge role in reducing the query latency, by determining the sub-query execution order in case of complex queries. The statistics about the number of triples in each sub-class can be used to build a frequency histogram. This table can be also modified dynamically by updating it based on the cardinality of triples with the most frequent attribute combinations, but that is beyond the scope of this research.

5. EXPERIMENTS

For the experiments, we used the LUBM [4] benchmark data and queries for performance evaluation and SP2 benchmark [15] for analyzing the query patterns. LUBM benchmark contains data that describes universities, students and the activities related to them, with a set of 14 benchmark queries to assess various performance criteria such as input data size, selectivity, and complexity. However like most works, we have also removed the OWL reasoning in queries by replacing the main-class with sub-class. For example, we replaced `ub:Student` with `ub:UndergraduateStudent` and `ub:Professor` with `ub:FullProfessor`.

Our experimental setup comprises a variable number of nodes (fifteen nodes is the default), each with a 16-core E5620 Intel Xeon CPU@2.40 GHz, 24 GB of RAM and is scalable to large data sets. We utilize the LUBM data sets: LUBM-1000, LUBM-2000 and LUBM-5000 which contains OWL files for 1000, 2000 and 5000 universities, respectively, with number of triples varying from: 140,000,000, 300,000,000 and 700,000,000 triples, and RDF file size in .NT format ranging from 24.1 GB to 118.8 GB.

6. PERFORMANCE EVALUATION

6.1 Data-Load Time

Figure 14 illustrates the data-load time comparison of JARS with various systems. JARS has a better data-load time accounting to the parallel batch insertion of data into all nodes. It takes approximately 6 hours for JARS to load the LUBM-5000 data which includes the time for RDF triple parsing, format conversion and dual-hash partitioning. As well as, it should be noted that twice the data is inserted into JARS accounting to the triple-replication. Since we

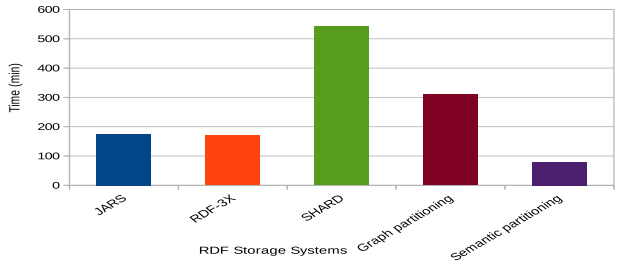

Figure 14: Data-Load Time comparison

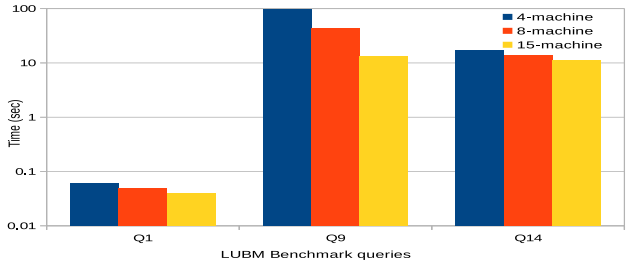

Figure 15: Scalability - By varying cluster size

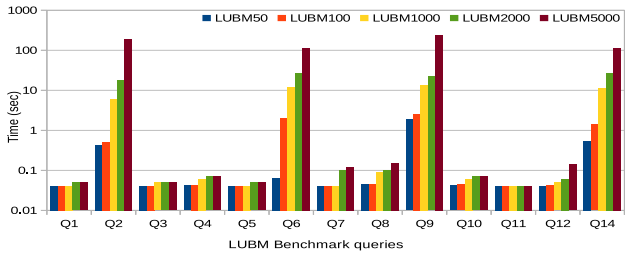

Figure 16: Scalability - By varying data size

avoid predicate-based hashing, we achieve better load balancing and the Relative Standard Deviation% of the number of triples in each node for the LUBM-5000 data is 0.04% for the subject tables and 46% for object tables.

6.2 Scalability

We measured the scalability of our system by comparing the query performance of LUBM queries on varying data sizes and varying cluster sizes.

Varying cluster size.

The query performance by varying the size of cluster with 4, 8 and 15 computing nodes over LUBM-1000 is illustrated in Figure 15. For a better understanding, we use three query varieties, a fast query, Q1, a complex query, Q9 and a data-intensive query, Q14. The performance of queries that use index-look ups and with smaller result size, such a Q1 and Q3, remains same irrespective of the cluster size. Also, in the case of 1-BGP queries with large input such as Q6 and Q14, adding more number of machines does not improve the performance, since our distribution is in such a way that all candidate results are on the same node. The complex queries such as Q2 and Q9 require inter-node communication and JARS could successfully make use of the number of computing nodes and the I/O bandwidth available in the cluster to improve the query response time. Scaling out the cluster causes the data set in each machine to be small enough and thus reduces the disk I/O. Therefore, cluster size has a good impact on complex queries rather than simple 1-BGP and 2-BGP queries.

Varying data sizes.

Figure 16 shows the query-response time over LUBM queries for varying data sizes, LUBM-50, LUBM-100, LUBM-1000, LUBM-2000 and LUBM-5000, using a 15-machine cluster. In most of the queries, the query-response time increases as the data size increases except for certain fast queries such as Q1 and Q3, which require only index lookups.

6.3 Query Performance Comparisons

For a better peformance analysis, we compare JARS with a centralized system, RDF-3X and some existing distributed systems which use similar hash partitioning strategies such as SHARD with simple hash partitioning technique, 1-hop and 2-hop graph partitioning techniques implemented in [6] and semantic hash partitioning in [9]. These works are briefly explained in 2 and are similar to JARS in their data partitioning strategies and utilization of clustered indexes. The query-response time comparison results with RDF-3X are shown in Figure 17. JARS exhibits various orders of magnitude better performance than RDF-3X for majority of the queries.

The query-response time comparisons with [6], [9] and SHARD are performed on LUBM-2000, illustrated in Figure 18. LUBM-2000 is used, since the compared systems use graph partitioner, METIS which crashes for large data sets like LUBM-5000. We have selected a subset of simple, complex and data intensive queries such as Q2, Q4, Q9, Q11 and Q14 for comparisons. JARS out-performs other systems with orders of magnitude on majority of the queries. For complex queries, JARS has only a comparable performance due to the powerful data processing platforms like Hadoop in other systems. We have also selected SHARD for our comparison because, it helps to compare the performance of our dual-hash partitioning method against the simple hash partitioning technique employed in SHARD. However, SHARD's hash partitioning technique only helps to optimize the subject-subject joins.

From these results, we deduce that JARS outperforms many of the relational state-of-the-art RDF storage approaches for all the common star - and chain-pattern SPARQL queries, irrespective of the selectivity and data intensiveness. The query-response time results for LUBM-5000 is illustrated in Table 1.

Unlike the other storage approaches, the maintenance cost due to data updates is significantly low in JARS. Addition/deletion of data does not require any modification in the relational schema like property table and vertical partitioning approach. The high cost incurred for data preprocessing and updates in graphical approach pose a serious setback to it's maintenance.

6.4 Discussion

We have conducted some preliminary evaluations on the required disk space consumption after RDF compression, by mapping the RDF URIs to unique IDs. The on-disk cumulative size, with the triple duplication and indexes, is

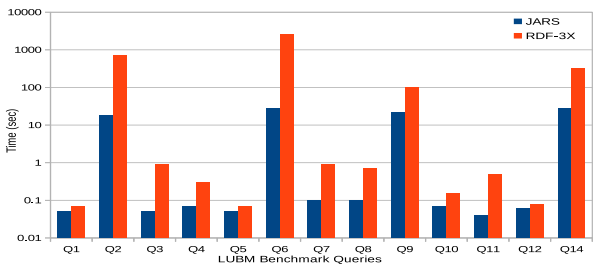

Figure 17: Comparison with RDF-3X

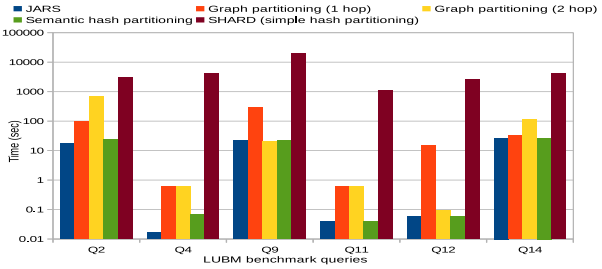

Figure 18: Comparison with distributed systems

Table 1: Performance on LUBM-5000 in seconds

Query-Response Time												
Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q14
0.05	195	0.05	0.07	0.05	115	0.12	0.15	245	0.07	0.04	0.14	113.5

approximately 355 GB for LUBM-5000, 145 GB for LUBM-2000 and 80 GB for LUBM-1000. Even though, there is further scope for compression, considering the reduced cost of storage, we are currently ignoring the space cost.

7. CONCLUSION AND FUTURE WORK

This paper presents JARS, a relational join-aware distributed RDF storage system, which co-locates join-able RDF triples through a novel dual-hash partitioning and selective clustering approach. JARS eliminates any inter-node communication in the processing of star- and simple chain-pattern SPARQL queries and mitigates the inter-node communication required for composite-pattern SPARQL queries, thereby improving the overall query-response time. We measured the data-load time, query-response time and evaluated the scalability of our system through extensive experiments, which indicates that JARS out-performs many of the existing RDF storage systems, in terms of query-response time, preprocessing and maintenance cost.

As a future work, we plan to make our system more space-efficient through compression of repeating RDF resources due to co-located triples and thus, maintain a balance between space efficiency and query efficiency by reducing the storage cost for indexes. Also, we hope that performance of JARS will improve on replacing with a better hashing algorithm on top of fast query processing platforms like Hadoop.

8. ACKNOWLEDGEMENTS

This research was supported in part by a JSPS Grant-in-Aid for Scientific Research (A) (#25240014).

9. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[2] C. Bizer, T. Heath, K. Idehen, and T. Berners-Lee. Linked data on the web (ldow2008). In *Proceedings of the 17th international conference on World Wide Web*, pages 1265–1266. ACM, 2008.

[3] O. Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[4] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[5] A. Holmes. *Hadoop in practice*. Manning Publications Co., 2012.

[6] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.

[7] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *Knowledge and Data Engineering, IEEE Transactions on*, 23(9):1312–1327, 2011.

[8] Jena, 2011. https://jena.apache.org/.

[9] K. Lee and L. Liu. Scaling queries over big rdf graphs with semantic hash partitioning. *Proceedings of the VLDB Endowment*, 6(14):1894–1905, 2013.

[10] Md5, 1992. https://en.wikipedia.org/wiki/MD5.

[11] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, 2010.

[12] A. Owens, A. Seaborne, N. Gibbins, et al. Clustered tdb: a clustered triple store for jena. 2008.

[13] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, page 4. ACM, 2010.

[14] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Spˆ 2bench: a sparql performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 222–233. IEEE, 2009.

[15] Sparql, 2008. https://en.wikipedia.org/wiki/SPARQL.

[16] X. Wang, T. Yang, J. Chen, L. He, and X. Du. Rdf partitioning for scalable sparql query processing. *Frontiers of Computer Science*, 9(6):919–933, 2015.

[17] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.