INVESTIGATION FOR DATA STORAGE AND QUERY PROCESSING IN MAP-REDUCE USING ALLEGROGRAPH, ORACLE 12C WITH PROPOSED ERDFMF

V.Shanmugapriya [1*], Dr. D. Maruthanayagam[2]
[1]Research Scholar, Sri Vijay Vidyalaya College of Arts & Science, Dharmapuri, Tamilnadu, India
[2] Head/Professor, PG and Research Department of Computer Science, Sri Vijay Vidyalaya College of Arts & Science, Dharmapuri, Tamilnadu, India

Abstract: Nowadays, a leading instance of big data is represented by Web data that lead to the definition of so-called big Web data. Indeed, extending beyond to a large number of critical applications (e.g., Web advertisement), these data expose several characteristics that clearly adhere to the well-known 3V properties (i.e., volume, velocity, variety). The Resource Description Framework (RDF) represents a main ingredient and data representation format for Linked Data and the big web data. It supports a generic graph-based data model and data representation format for describing things, including their relationships with other things. As the size of RDF datasets is growing fast, RDF data management systems must be able to cope with growing amounts of data. Even though physical dealing with RDF data using a relational desk is possible, querying a giant triple table turns into very high-priced due to the fact of the a couple of nested joins required for answering layout queries. When RDF graphs are defined on pinnacle of big (Web) data, they lead to the so-called large-scale RDF graphs, which fairly populate the next-generation Semantic Web. In order to manner such type of big data, MapReduce, an open source computational framework especially tailored to big data processing, has emerged throughout the ultimate years as the reference implementation for this quintessential setting***. In this paper, presents the proposed ERDFMF (Efficient RDF Process the usage of MapReduce Framework), a scalable and environment friendly RDF triple keep and a SPARQL question optimization approach*** for Big RDF information on a distributed Hadoop based totally RDF system. ***Experiments exhibit that ERDFMF outperforms better than usual existing disbursed RDF triple stores,*** with remarkable scalability and fault tolerance.

Keywords: ***Big data, Resource Description Framework, MapReduce Framework, AllegroGraph, Oracle 12c, Query Execution algorithm, Scalability and Fault Tolerance***

I.INTRODUCTION

Today the resources and services available in the web are growing at a fast way day by day, this massive data considered unstructured, distributed and heterogeneous presents a problem in the time of data analysis, access and maintain relevant resources for a user query. It is difficult to find a solution for the big data storage and its access and visualization issue. There is a need to develop solutions to manage large amounts of data on a daily basis and extract new knowledge from them [1]. Big data storage methods are different from traditional storage methods. There are critical factors/features that should be considered for storage: consistency (C), availability (A), and partition tolerance (P) [2, 3]. An important aspect in efficient data management is making data interoperable enabling the of extract meaningful information from large datasets. To this end, the Semantic Web technologies proposed by Tim Berners Lee, offers a remedy to interlink highly distributed data and enhancing data interoperability, to share and reuse data between multiple applications and help users to extract new knowledge. Semantic knowledge representation is utilized in many domains, especially in those with complex information management requirements. Domains such as health care, manufacturing, defense industries, commerce, law [4], government, etc. all have very different data modeling requirements and very complex information management issues. In this context, the W3C recommends the RDF formalism to describe the resources semantically, however, only a format of resource representation is not enough and we need RDF resource query languages called SPARQL [5]. Therefore for handling and analysis big data, we need to use these languages as a first rule for efficient storing and query processing. Therefore, it will exploit the MapReduce for the parallels of the calculation of the SPARQL query, and the storage maintenance of the global RDF graph in HDFS. In this paper, propose an efficient RDF storage and SPARQL query processing framework for large-scale RDF datasets via Map Reduce. **For experiments, use the LUBM and SP2B data set.** It is a **benchmark data set designed to enable researchers to evaluate a big RDF data.** The LUBM and SP2B data generator generates data in RDF/XML serialization format [6]. This format is not suitable for proposed system purpose because we store data in HDFS as at files and so to retrieve even a single triple we would need to parse the entire file. ***Therefore we reduce RDF graph using bisimulation reduction*** [13,14] then convert the RDF data to N-Triples to store the data, because with that format we have a complete RDF triple (Subject, Predicate and Object) in one line of a file, which is very convenient to use with MapReduce jobs.

For querying process, ERDFMF advocates the reliance on workload guided partitioning, where data is incrementally and dynamically repartitioned based on the workload. However, at any moment in time, there might be queries that are not favored by the current distribution. ***While ERDFMF will eventually adapt to them,*** these ***queries*** need to be ***executed efficiently; otherwise, the whole system performance will degrade.*** Therefore, ERDFMF introduces an efficient baseline for distributed SPARQL query evaluation. ERDFMF exploits the hash-based data locality to execute queries comparable or faster than state-of-the-art distributed RDF systems.

II. RDF STORES

As the storing, accessing, and processing of large amount of data represents an essential requirement in order to satisfy the needs to search, analyze and visualize these data as information, the capability to manage a large number of triples (greater than 2 billions) has been chosen as filter criterion for selecting a set of stores to be evaluated [7, 8]. Based on this filter, it has been selected a list among the most well-known RDF stores that satisfy this requirement, excluding all technologies which are discontinued or in a too early stage of development. The number of triple stores has been considerably increased from Jena and Sesame in the early 2000s to YARS2, Jena TDB, Jena SDB, Virtuoso, AllegroGraph, BigData, Mu lgara, Sesame, Kowari, 3Store and RDF Gateway. Among these some like Garlik and YARS2 are not distributed [9]. A few like AllegroGraph are commercially available. The others are open sources. Majority of the efforts were laid in to check on the freely available open source triple stores and hence the choice of Allegrograph, Sesame, and Jena API.

2.1 Allegrograph

The existing system producers of Allegrograph are Franz's Semantic Technology solutions Company. It is a persistent storage system with their own implementation of databases. Figure 1 illustrates an architectural overview of Allegrograph [10] and the features are as follows.

- It supports transactions such as Commit, Rollback, check pointing.
- It has ability to recover data fast.
- It provides 100% read Concurrency, and full write concurrency
- Ensures dynamic/auto indexing
- Powerful and expressive reasoning/querying.

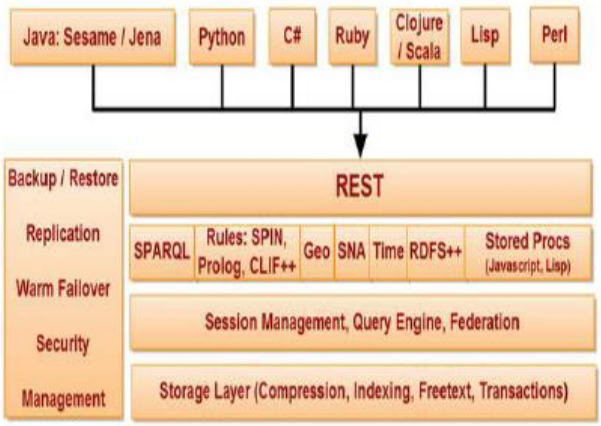

Figure 1: Allegrograph Architecture [10].

AllegroGraph provides REST protocol architecture, truly a superset of the Sesame HTTP Client. Franz's staff immediately supports adapters for a variety of languages, Sesame Java, Sesame Jena, Python the usage of the Sesame signatures, and Lisp. **AllegroGraph is a modern, high-performance, power graph database**. AllegroGraph uses efficient memory utilization in aggregate with disk-based storage, enabling it to scale to billions of quads while maintaining most advantageous overall performance [11]. AllegroGraph's SPARQL, one of the W3C's "interoperable implementations", consists of a query optimizer, and has full support for named graphs. ***It additionally offers ACID transaction support; ACID*** (Atomicity, Consistency, Isolation, and Durability) is a set of properties that assurances that database transactions are processed reliably.

2.1 Oracle 12c

Oracle 12c [12], which was released in June 2013, is used. The semantic residences of the Oracle 12c database and their relations are proven in Figure 2. As viewed in Figure 2, **Oracle 12c can incorporate each semantic and relational data at the same time** [15]. DB-Engines collects and presents data on database management systems, and also ranks them in extraordinary lists in accordance to the scope of usage such as RDF store, relational DBMS, key-value store, report store, etc. According to DB-Engines, Oracle is the most famous system for relational DBMS and in overall database management systems [16]. Oracle is the first relational database administration device that supports RDF storage. Therefore, the usage of Oracle as a RDF shop simplifies the integration of present normal statistics with triples. The efficient way to load semantic data is bulk loading. In this analysis, bulk loading is used to insert semantic records from ontologies into Oracle 12c triple store. However, semantic data can additionally be added triple by way of triple by way of the use of the INSERT declaration in SPARQL. ***In Oracle 12c, semantic data can be queried efficiently*** [17]. Additionally, relational and semantic records can be queried by way of ability of ontologies in order to discover relations between relational and semantic data. After loading semantic data, the energy of querying semantic records can be expanded by using the use of regulations and inference engines [18, 19]. Inference gives logical results with regards to statistics and rules.

However, in this study, the inferred triples are no longer evaluated in order to now not make the assessment greater complex.

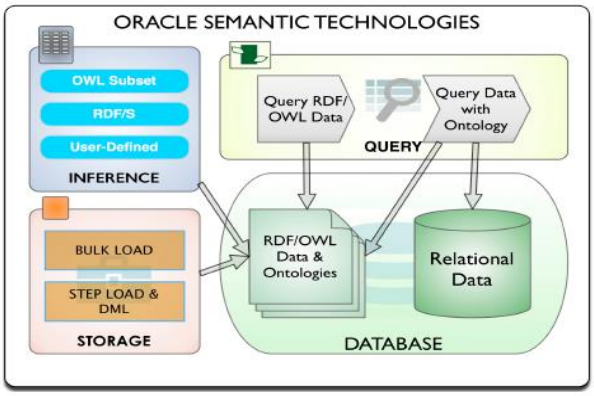

Figure 2: The semantic properties of Oracle 12c Triple Store.

III. PROPOSED SYSTEM ARCHITECTURE

This section describes the architecture of proposed framework. The framework consists of **two components.** The **upper part** of Figure 3 depicts the data preprocessing component and the **lower part** shows the query answering one.

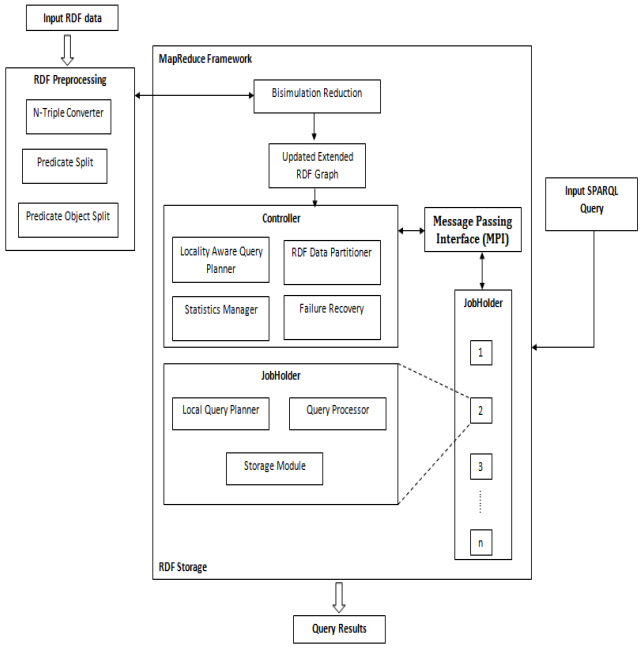

Figure 3: Architecture of Proposed ERDFMF

A. DATA PREPROCESSING COMPONENT

It has **four subcomponents** for data generation and preprocessing. 1. Bisimulation reduction to reduce the size of an RDF graph while keeping its inherent structural properties and provide a scalable implementation based on MapReduce. 2. Convert RDF/XML to N-Triples serialization format using N-Triples Converter component. 3. The Predicate Split (PS) component takes the N-Triples data and splits it into predicate files. 4. The predicate files are then fed into the Predicate Object Split (POS) component which splits the predicate files into smaller files based on the type of objects.

1. Compaction of RDF Graphs based on Bisimulation Reduction

Proposed algorithm to compute a bisimulation reduction of an RDF graph G is based on the naive bisimulation method. Two MapReduce jobs are created for each iteration of the bisimulation reduction algorithm. The ***first one, called update job,*** computes the new signature and updates the ID for each node while the ***second one, called counting job***, counts the number of distinct IDs. The program terminates if the number of distinct IDs does not change in two consecutive counting jobs, i.e. no blocks are split anymore. In order to compute the signature of a node s, need all triples (i.e. edges) where s is the subject, $s \xrightarrow{p} o$, as well as the current block ID of the object o, ID(o). To get both at the same time, store ID (o) in every triple where o is the object, i.e. (s, p, o, ID (o)), and call such an extended triple a quad; accordingly, RDF graphs are now defined over quads and call it an extended RDF graph. After computing the signature of a node s, have to update the ID of every quad where s is the object, i.e. (_,_, s, ID(s)), so that this updated ID can be used in the next iteration. Consequently, also need those quads with s as object, have to update the ID of s in these quads after the signature computation of s is complete. In each iteration, the update job receives the extended RDF graph as input and outputs the graph with updated IDs. There is no extra data preprocessing step necessary, since the algorithm starts with all nodes lying in the same block. Hence, use the same default ID for every node in the first iteration.

Algorithm 1: Update Job - map(key, value)

input : key: byte offset in input file, can be ignored
value: a quad (s, p, o, $ID^k(o)$)

1. (s, p, o, $ID^k(o)$) parseQuad(value)
2. emit((s,0), (s, p, o, $ID^k(o)$))
3. emit((o,1), (s, p, o, $ID^k(o)$))

Above algorithm 1 outlines the map function of the update job. It receives a quad from the extended RDF graph as input. After parsing the quad, two output key value- pairs are written which both have the input quad as value. The key is a custom composite-key consisting of the subject or the object, respectively, and a number indicating whether the first part of the key was the subject (0) or the object (1) in the input quad. In the following Shuffle & Sort phase, each reducer then receives all quads of a node s where s is either subject or object. These triples are sorted by the second part of the composite key using the secondary sort functionality of Hadoop, resulting in all quads with s as subject being sorted before those with s as object. This enables us to first compute the new signature and ID of s and then output the quads with updated IDs within the same reduce call.

Algorithm 2: Update Job - reduce(key, values)

input : key: composite key (node s, sortOrder)
values: a list of quads [(s, p, o, $ID^k(o)$)]

1. $sig^{k+1}(s) \leftarrow \{\}$
2. while value in values ^ key.sortOrder = 0 do
3. (s, p, o, $ID^k(o)$) parseQuad(value)
4. $sig^{k+1}(s)$ $sig^{k+1}(s) \cup \{(p, ID^k(o))\}$
5. end

6. $ID^{k+1}(s) \leftarrow hash(sig^{k+1}(s))$
7. while value in values ^ key.sortOrder = 1 do
8. $(x, p, s, ID^{k}(s))$ parseTriple(value)
9. emit("graph", $(x, p, s, ID^{k+1}(s))$)
10. end
11. emit("node IDs", $(s, ID^{k+1}(s))$)

Algorithm 2 shows the reduce function of the update job which uses the discussed ordering of quads to first compute the signature and ID of a node *s*. If the key is subject of the quad, the predicate and current ID of the object is added to the signature of *s*. Once the signature is complete, the new ID of *s* is calculated and used to update the ID in the remaining quads where *s* is the object. It is crucial that the hash function outputs the same value for two identical sets of elements independently of the order - or alternatively the set must be sorted before hashing. The following counting job counts all distinct IDs and compares it with the previous count. There may be nodes that do not have incoming edges and hence do not appear as an object in a quad. These nodes are not necessary for calculating the signatures of other nodes, however their IDs need to be considered in the counting job to get the correct number of distinct IDs. Therefore, every node and its ID are stored in a separate file called "node IDs" in HDFS, while the updated extended RDF graph is stored in the file "graph". The node IDs are used as input for the counting job, whereas the graph is used for consecutive iterations. This format is not suitable for proposed purpose because the store data in HDFS as at files and so to retrieve even a single triple would need to parse the entire file. Therefore convert the data to N-Triples to store the data, because with that format have a complete RDF triple (Subject, Predicate and Object) in one line of a file, which is very convenient to use with MapReduce jobs. The processing steps to go through to get the data into the intended format are described in following steps.

2. Predicate Split (PS)

In this step, again divide the updated extended RDF graph stored in graph file data according to the predicates. This division immediately enables us to cut down the search space for any SPARQL query which does not have a variable predicate. For such a query, just pick a file for each predicate and run the query on those files only. For simplicity, name the files with predicates, e.g. all the triples containing a predicate p1: pred go into a file named p1_pred. However, in case have a variable predicate in a triple pattern and if cannot determine the type of the object, have to consider all files. If determine the type of the object then consider all files having that type of object. In real world RDF data sets, the number of distinct predicates is in general not a large number. However, there are data sets having many predicates. Proposed system performance does not vary in such a case because select files related to the predicates specified in a SPARQL query.

3. Predicate Object Split (POS)

- **Split Using Explicit Type Information of Object:** In the next step, work with the explicit type information in the rdf_type file. The predicate rdf:type is used in RDF to denote that a resource is an instance of a class. The rdf_type file is first divided into as many files as the number of distinct objects the rdf:type predicate has. For example, if in the ontology the leaves of the class hierarchy are c_1, c_2,……,c_n then will create files for each of these leaves and the file names will be like type_c_1, type_c_2, ... , type_c_n. Please note that the object values c_1, c_2,……,c_n are no longer needed to be stored within the file as they can be easily retrieved from the file name. This further reduces the amount of space needed to store the data. Generate such a file for each distinct object value of the predicate rdf:type.
- **Split Using Implicit Type Information of Object:** Divide the remaining predicate files according to the type of the objects. Not all the objects are URIs, some are literals. The literals remain in the file named by the predicate; no further processing is required for them. The type information of a URI object is not mentioned in these files but they can be retrieved from the type_* files. The URI objects move into their respective file named as predicate type. For example, if a triple has the predicate p and the type of the URI object is c_i, then the subject and object appears in one line in the file p_c_i. To do this split, need to join a predicate file with the type_* files to retrieve the type information.

B. QUERY ANSWERING COMPONENTS

Proposed system exploits the hash-based data locality to execute queries comparable or faster than state-of-the-art distributed RDF systems. Proposed system employs the typical master-slave paradigm. Proposed system uses the standard Message Passing Interface **(MPI)** for master-worker communication. The master begins by encoding the data and partitioning it among workers. Each worker loads its triples and collects local statistics. Then, ***the master aggregates these statistics and becomes ready for answering queries.*** Each query is submitted to the master, which decides whether the query can be executed in parallel or distributed mode. In parallel mode, the query is evaluated concurrently by all workers without communication. Queries in distributed mode are also evaluated by all workers but require communication.

1. Master

- **String Dictionary:** RDF data contain long strings in the form of URIs and literals. To avoid the storage, processing, and communication overheads, encode RDF strings into numerical IDs and build a bi-directional dictionary.
- **Data Partitioner:** ***Proposed system uses lightweight node based partitioning using subject values.*** Given W workers, a triple t is assigned to worker w_i, where i is the result of a hash function applied on t.subject. This way all triples that share the same subject go to the same worker. Consequently, any star query joining on subjects can be evaluated without communication cost. Proposed system does not hash on objects because they can be literals and common types; this would assign all

triples of the same type to one worker, resulting in load imbalance and limited parallelism.

- **Statistics Manager:** It maintains statistics about the RDF data, which are used for global query planning and during adaptivity. Statistics are collected in a distributed manner during bootstrapping.
- **Locality-Aware Query Planner:** Planner uses the global statistics from the statistics manager and the pattern index from the redistribution controller to decide if a query, in whole or partially, can be processed without communication. Queries that can be fully answered without communication are planned and executed by each worker independently. On the other hand, for queries that require communication, the planner exploits the hash-based data locality and the query structure to find a plan that minimizes communication and the number of distributed joins.
- **Failure Recovery:** The master does not store any data but can be ***considered as a single-point of failure*** because it maintains the dictionaries, global statistics, and Passing Interface. **A standard failure recovery mechanism can be employed by proposed system**. Assuming stable storage, the master can recover by loading the dictionaries and global statistics because they are read-only and do not change in the system. The Passing Interface can be recovered by reading the query log and reconstructing the heat map. Workers on the other hand store data; hence, in case of a failure, data partitions need to be recovered. A fast failure recovery solution for distributed graph processing systems is a hybrid of **checkpoint-based and log-based recovery schemes.** This approach can be used by proposed to recover worker partitions and reconstruct the replica index.

2. Worker

- **Storage Module:** Each worker w_i stores its local set of triples D_i in an in-memory data structure, which supports the following search operations, where s, p, and o are subject, predicate, and object, respectively: 1. given p, return set $\{(s, o) \mid \langle s,p,o\rangle \in D_i\}$. 2. given s and p, return set $\{o \mid \langle s,p,o\rangle \in D_i\}$. 3. given o and p, return set $\{s \mid \langle s,p,o\rangle \in D_i\}$. Since all the above searches require a known predicate, primarily hash triples in each worker by predicate. The resulting predicate index (simply P-index) immediately supports search by predicate (i.e., the first operation). Furthermore, use two hash maps to re-partition each bucket of triples having the same predicate, based on their subjects and objects, respectively. These two hash maps support the second and third search operation and they are called predicate-subject index (PS-index) and predicate-object index (PO-index), respectively. Given the number of unique predicates are typically small; the storage scheme avoids unnecessary repetitions of predicate values. Note that when answering a query, if the predicate itself is a variable, then simply iterate over all predicates. Indexing scheme is tailored for typical RDF knowledge bases and their workloads. Being orthogonal to the rest of the system, alternative schemes, like indexing all SPO combinations, could be used at each worker. Finally, the storage module computes statistics about its local data and shares them with the master after data loading.
- **Query Processor:** Each worker has a query processor that operates in two modes: (i) Distributed Mode for queries that require communication. In this case, the locality aware planner of the master devises a global query plan. Each worker gets a copy of this plan and evaluates the query accordingly. Workers solve the query concurrently and exchange intermediate results (ii) Parallel Mode for queries that can be answered without communication. In this case, the master broadcasts the query to all workers. Each worker has all the data needed for query evaluation; therefore it generates a local query plan using its local statistics and executes the query without communication.
- **Local Query Planner:** ***Queries executed in parallel mode are planned by workers autonomously***. For example, star queries joining on the subject are processed in parallel due to the initial partitioning.

Based on the above scenarios, introduce Locality-Aware Distributed Query Execution algorithm (see Algorithm 3). The algorithm receives an ordering of the subquery patterns. For each join iteration, if the second subquery joins on the pinned subject, the join is executed without communication (line 7). Otherwise, the join is evaluated by the Distributed selective join algorithm (lines 8-28). In the first iteration, p_1 is a base subquery pattern; however, for the subsequent iterations, p_1 is a pattern of intermediate results. If p_1 is the first subquery to be matched, each worker finds the local matching of p_1 (line 10) and projects on the join column c1 (line 13). If the join column of q_2 is subject, then each worker hash distributes the projected column (line 15); or sends it to all other workers otherwise (line 17). To avoid the overhead of synchronization, communication is carried out using non-blocking MPI routines. All workers perform semi-join on the received data (line 22) and send the results back to w (line 23). Finally, each worker finalizes the join (line 27) and formulates the final result (line 28). Lines 22 and 27 are implemented as local hash-joins using the local index in each worker. The result of Distributed selective join iteration becomes p_1 in the next iteration.

Algorithm 3: Input: Query Q with n ordered subqueries $\{q_1, q_2, \ldots q_n\}$

Result: Answer of Q

1. $p_1 \leftarrow q_1$;
2. pinned_subject $\leftarrow p_1$.subject;
3. for i $\leftarrow$2 to n do
4. $p_2 \leftarrow q_i$;
5. $[c_1, c_2]$ getJoinColumns(p_1, p_2);
6. if c_2 == pinned_subject AND c_2 is subject then
7. $p_1 \leftarrow$ JoinWithoutCommunication (p_1, p_2, c_1, c_2);
8. else
9. if p_1 NOT intermediate pattern then
10. RS_1 answerSubquery(p_1);
11. else
12. RS_1 is the result of the previous join
13. $RS_1[c_1] \leftarrow \pi_{c1}$ (RS_1); // projection on c_1
14. if c_2 is subject then

15. Hash $RS_1[c_1]$ among workers;
16. else
17. Send $RS_1[c_1]$ to all workers;
18. Let RS_2 answerSubquery(p_2);
19. foreach worker w, w : 1 → N do
20. $RS_{1w}[c_1]$ is the $RS_1[c_1]$ received from w
21. CRS_{2w} are candidate triples of RS_2 that join with $RS_{1w}[c_1]$
22. $CRS_{2w} \leftarrow RS_{1w}[c_1] \rtimes RS_{1w}[c_1].c_1=RS_2.c_2\ RS_2$;
23. Send CRS_{2w} to worker w;
24. foreach worker w, w : 1 → N do
25. $RS2_w$ is the CRS_{2w} received from worker w
26. RES_w is the result of joining with worker w
27. $RES_w \leftarrow RS_1 \bowtie RS_1.c_1=RS_2w.c_2\ RS_2w$;
28. p1← $RES_1 \cup RES_2 \cup .... \cup RES_N$;

IV. EXPREMENTAL RESULTS AND DISCUSSIONS

In this section, **evaluate the performance of proposed architecture (ERDFMF).** ERDFMF was implemented using java. We used hadoop- 2.7.1 for proposed ERDFMF framework. Install and run the software with the system configuration specified in: Intel Xeon CPU E5-2640 v4 @2.40GHz and 124 GB RAM. ***The proposed framework compared with Allegrograph and Oracle 12c.*** For comparison, also verify the performance of several typical distributed RDF triple stores under the same environment. In our experiments with SPARQL query processing, **use two synthetic data sets: LUBM and SP2B.** The LUBM data set generates data about universities by using an ontologyy. It has 14 standard queries. Some of the queries require inference to answer. The LUBM data set is very good for both inference and scalability testing. For all LUBM data sets, we used the default seed. The SP2B data set is good for scalability testing with complex queries and data access patterns. It has 16 queries most of which have complex structures.

Load Time

Figures 4 and 5 shows the graphical representation of the loading times. ERDFMF, AllegroGraph and Oracle 12c can be used to store similar data, but they have different loading times of the same data set. **Experimental results showed that ERDFMF had better load time and resource utilization than AllegroGraph and Oracle 12c** during bulk load due to their converting and RDF compaction for storing large size of data. AllegroGraph showed better performance for bulk load over other semantic web database Oracle 12c. On the contrary, AllegroGraph is loading and matching RDF/OWL data faster than Oracle 12c due to its design purpose. Oracle has a slow loading time on big data sets due to its layered infrastructure.

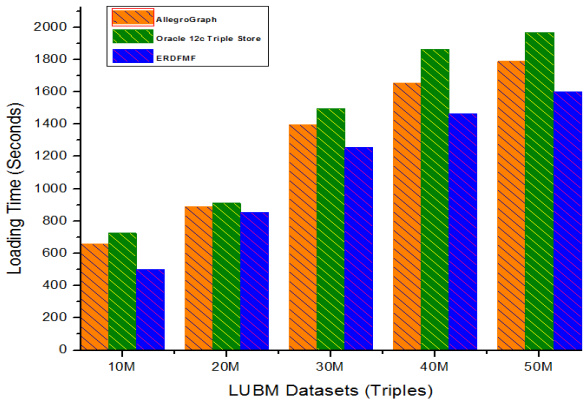

Figure 4: LUBM Dataset Load Time

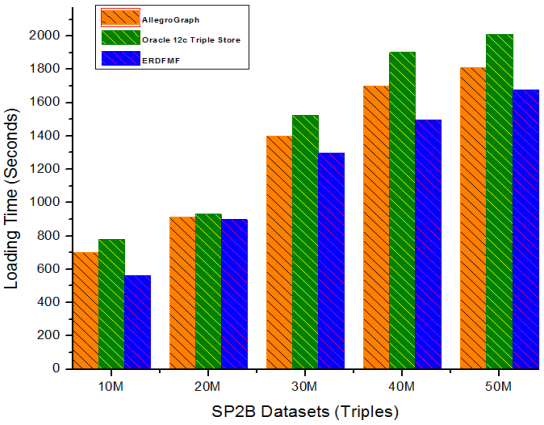

Figure 5: SP2B Dataset Load Time

Query Response Time

Performance comparison between ERDFMF, AllegroGraph and Oracle 12c storage schemes for LUBM queries 1 to 14 of 10 million and 20 million triples is shown in Figure 6 and Figure 7 shows the SP2B query response time of 10 and 20 million triples for 1 to 16 queries using ERDFMF, AllegroGraph and Oracle 12c storage schemes. Results of queries, implementing the LUBM, show that all stores failed to retrieve results for the large dataset size. ***ERDFMF showed better Query response time and scaled well for memory and CPU utilization than AllegroGrpah.*** ERDFMF and AllegroGrpah showed shorter QRT than Oracle 12c. In results for queries using SP2B, ERDFMF displayed better results for SP2B queries than others. AllegroGrpah shows better results for SP2B identification than Oracle 12c. Finally concluded, that ERDFMF outperforms AllegroGraph and Oracle 12c in query response time of queries from LUBM Q1 to Q14 and SP2B Q1 to Q16. Query response time depends on the size (number of rows) of each temporary view.

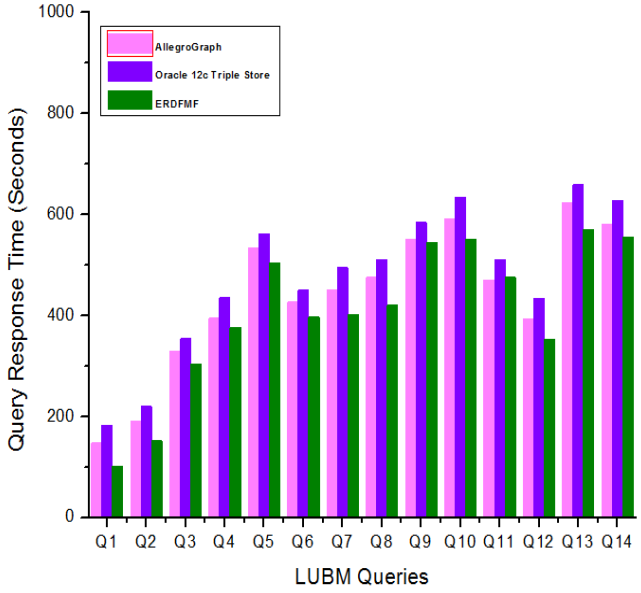

Figure 6: Response Time for LUBM Queries

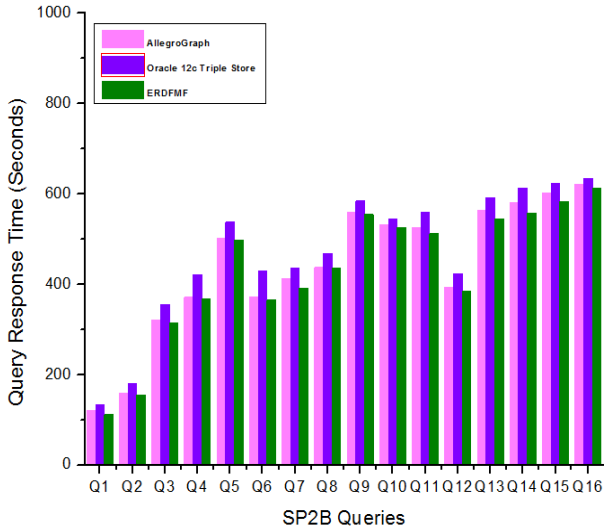

Figure 7: Response Time for SP2B Queries

Memory Consumption

Figure 8 to 11 reports the result of memory consumption. Note that, a part of the memory that is occupied by the operating system, the maximum available memory for the applications is always lower than the size of RAM. On the hardware device, the memory consumption of ERDFMF and AllegroGraph gradually increased according to the size of the storage while the buffering memory of Oracle 12c was statically set. In the throughput test (see Figure 8 and 9), the memory consumption of Oracle 12c rose up to 50 MB and 55 MB after inserting 2 million, 10 million respectively. Meanwhile, AllegroGraph required more than 40 MB to add more than 5 million triples and 50 for 20 million triples. That explains the lower scale of AllegroGraph and Oracle 12c comparing to ERDFMF engine. On the other hand, ERDFMF consumed 45 MB of memory at most even when the storage goes up to 30 million triples. **In the query evaluating tests, AllgreoGraph and ERDFMF used less memory** than they did in the update throughput tests. Even with the dataset of 30 million triples, ERDFMF used only 45 MB. That is only a half of memory that AllegroGraph used in the query test with 10 million triples and is one-third of the memory that the Oracle 12c constantly occupies.

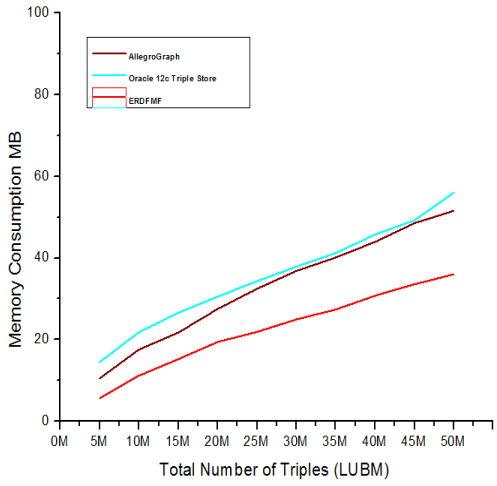

Figure 8: Memory Consumption for inserting (LUBM) of ERDFMF, AllegroGraph and Oracle 12c Triple Store

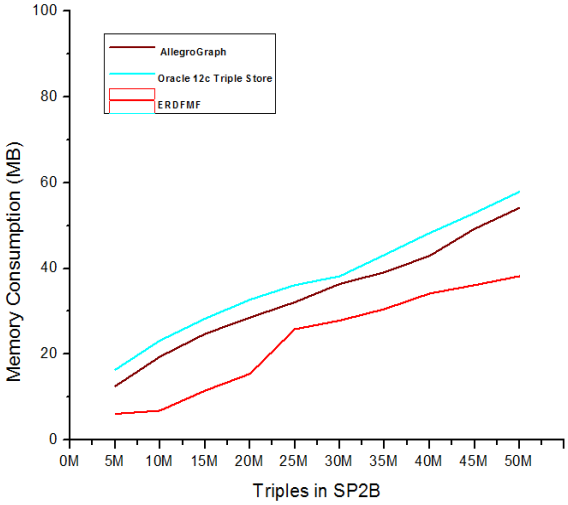

Figure 9: Memory Consumption for inserting (SP2B) of ERDFMF, AllegroGraph and Oracle 12c Triple Store

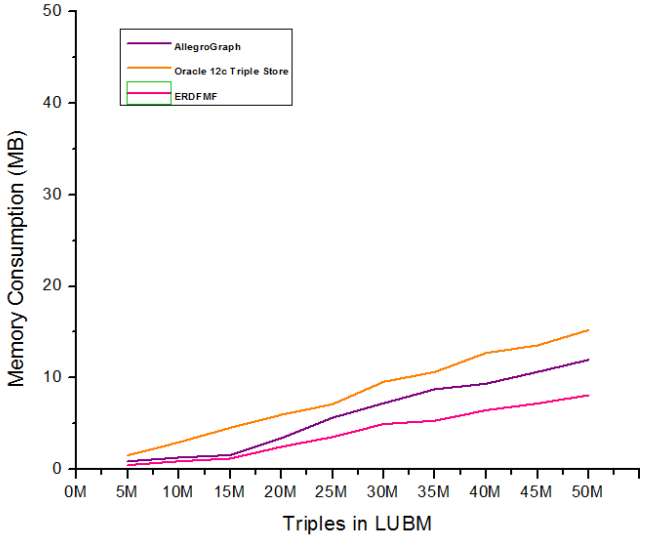

Figure 10: Memory Consumption for Querying (LUBM) of ERDFMF, AllegroGraph and Oracle 12c Triple Store

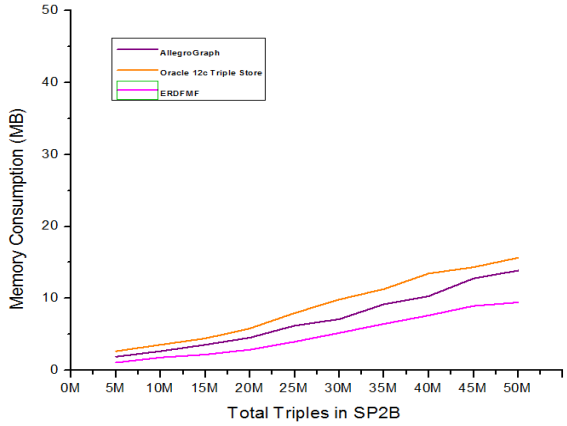

Figure 11: Memory Consumption for Querying (SP2B) of ERDFMF, AllegroGraph and Oracle 12c Triple Store

Scalability

In this graph, we show how the performance of different distributed systems change as we increase the dataset size. Using LUBM, we generated 5 datasets ranging from 100 million triples to 500 million triples; LUBM100, LUBM200, LUBM300, LUBM400, LUBM500. Similarly, we used SP2B data generator to create 5 datasets with sizes ranging from 100 to 500 million triples. Figures 12(a) and 12(b) show the throughput (queries per hour) of the selected systems for both LUBM and SP2B datasets. As expected, the throughput of most of the systems excluding Oracle 12c decrease slowly with the dataset size. **ERDFMF achieves the best throughput followed by AllegroGraph and Oracle 12c Triple Store.**

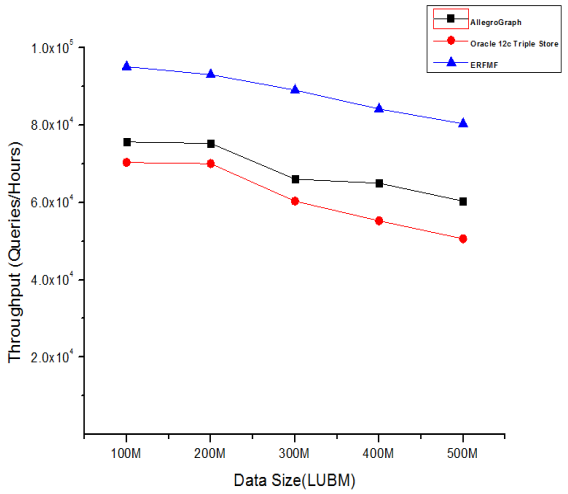

Figure 12(a) LUBM Scalability

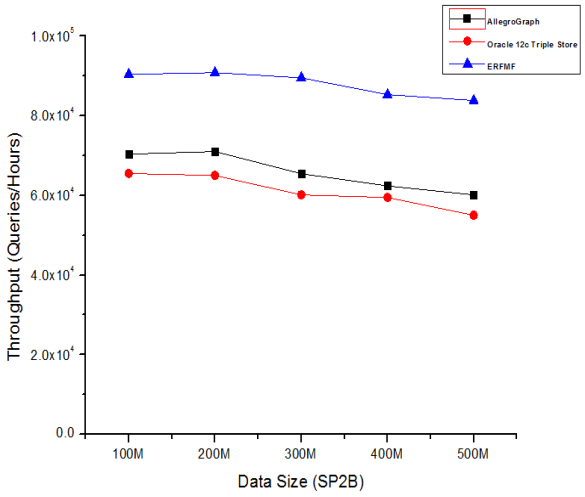

Figure 12(b) SP2B Scalability

Robustness

The robustness of each approach, evaluated in terms of percentage of unanswered queries within the stipulated time, is shown in Figure 13(a) and (b). For the given time constraint, Oracle 12c Triple Store are unable to output results for size 20 and 30 onwards respectively. Although AllegroGrpah output results until query size 50, their time performance is still poor. However, as the query size increases, the percentage of unanswered queries for both AllegroGraph and Oracle 12c keeps on increasing from ~0% to 65% and ~45% to 95% respectively. On the other hand ERDFMF answers >98% of the queries, even for queries of size 50, establishing its robustness. Analyzing the results for SP2B queries, **observe that in Figure 13(b), Oracle 12c are the slowest engines; AllegroGrpah perform better than them but nowhere close to ERDFMF**. We further observe that Oracle 12c are the least robust as they don't output results for size 30 onwards; on the other hand ERDFMF is the most robust engine as it answers >90% of the queries even for size 50. The percentage of unanswered queries for AllegroGraph and

Oracle 12c increase from 0% to ~70% and 80% to ~85% respectively, as we increase the size from 10 to 50.

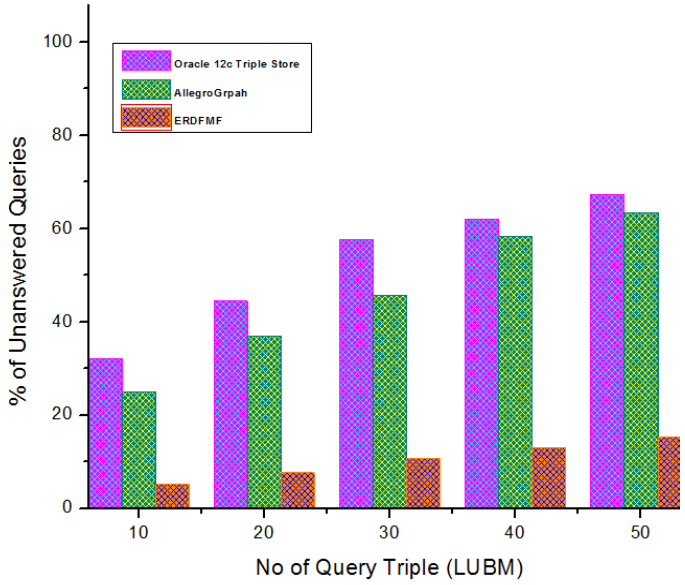

Figure 13 (a): Unanswered Queries (LUBM)

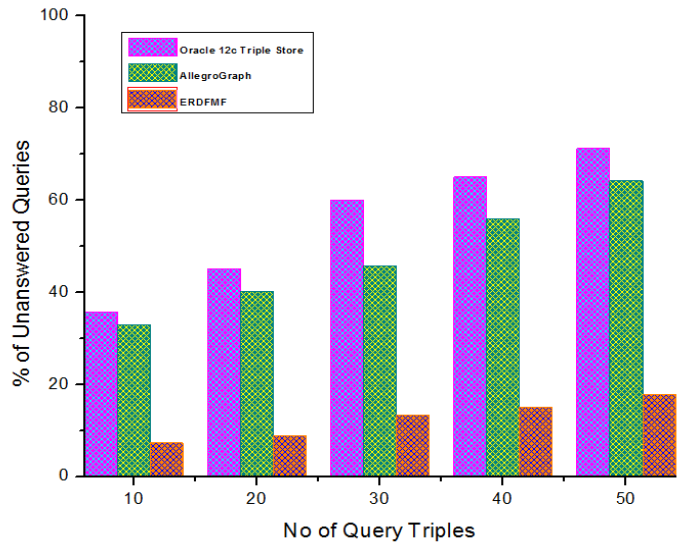

Figure 13 (b): Unanswered Queries (SP2B)

V. CONCLUSION

In this paper MapReduce based (ERDFMF) RDF engine, is introduced. **The proposed system aims to reduce the gap between the resource limitation of the targeted devices** and the scalability, robustness for an RDF engine. Considering the distinct nature of hardware of the devices, ERDFMF comes up with a flash-aware storage structure and an indexing scheme together with a low-memory-footprint join strategy. ERDFMF starts significantly fast by employing lightweight partitioning that hashes triples on the subjects. ERDFMF exploits query structures and the hash-based data locality in order to minimize the communication cost during query evaluation. By exploiting hash-based locality, **ERDFMF achieves better performance** than other pre-existing systems that employ sophisticated partitioning schemes. **ERDFMF** scales to very large RDF graphs and consistently **provides superior performance** by adapting to dynamically changing workloads. As a result, ERDFMF has the size significantly smaller, and ***faster performance in both LUBM and SP2B benchmark queries*** than those of generic RDF engines like AllegroGrpah and Oracle 12c.

VI. REFERNCES

[1]. S. Abburu and S. B. Golla. Effective partitioning and multiple RDF indexing for database triple store. Engineering Journal, 19(5):139–154, 2015.

[2]. I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. Proceedings of the VLDB Endowment, 10(13):2049–2060, 2017.

[3]. G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis, and D. Plexousakis. Incremental Data Partitioning of RDF Data in SPARK. In European SemanticWeb Conference, pages 50–54, Monterey, USA, 2018.

[4]. Microsoft, "The Big Bang: How the Big Data Explosion is Chaning the World", April 2013

[5]. 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 the 2013 international conference on Management of data. ACM, 2013, pp. 121–132.

[6]. Albert Haque, Lynette Perkins, "Distributed RDF Triple Store Using HBase and Hive", December 2012

[7]. Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed Mehdi-Reza Beheshti, and Sherif Sakr. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *Proc. VLDB Endow.*, 8(6): 654–665, 2015.

[8]. Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’10, pages 975–986, 2010.

[9]. http://www.bioontology.org/wiki/images/6/6a/Triple_Store s.pdf

[10]. http://www.franz.com/agraph/allegrograph

[11]. Oracle Spatial and Graph. Available online: http://www.oracle.com/technetwork/database/options/ spatialandgraph/overview/index.html (accessed on 1 December 2016).

[12]. Oracle Semantic Technologies Devepoler’s Guide 11g Release 2. Available online: http://docs.oracle.com/ cd/E11882_01/appdev.112/e25609/title.htm (accessed on 1 December 2016).

[13]. Stefan Blom and Simona Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *International Journal on Software Tools for Technology Transfer*, 7(1): 74–86, 2005.
[14]. Paris C. Kanellakis and Scott A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 228–240, New York, NY, USA, 1983. ACM
[15]. Park, S. (2013). Visualization of Resource Description Framework Ontology Using Hadoop, 228–231. doi:10.1109/IMIS.2013.46
[16]. Shen, Yanyan and Chen, Gang and Jagadish, H. V. and Lu, Wei and Ooi, Beng Chin and Tudor, Bogdan Marius, "Fast Failure Recovery in Distributed Graph Processing Systems," PVLDB, vol. 8, no. 4, 2014.
[17]. Neumann, Thomas and Weikum, Gerhard, "RDF-3X: A RISC-style Engine for RDF," PVLDB, vol. 1, no. 1, pp. 647–659, 2008.
[18]. Weiss, Cathrin and Karras, Panagiotis and Bernstein, Abraham, "Hexastore: Sextuple Indexing for Semantic Web Data Management," PVLDB, vol. 1, no. 1, pp. 1008–1019, 2008.
[19]. (Online Resource) The Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/

ABOUT THE AUTHORS

V.Shanmugapriya received her **M.Phil** Degree from Periyar University, Salem in the year 2007. She has received her **M.C.A** Degree from Madurai Kamaraj University, Madurai in the year 2002. She is working as Assistant Professor, Department of Computer Science, **PGP** College of Arts & Science, Namakkal, Tamilnadu, India. She has around **16** years of experience in Academic Field. She is pursuing her Ph.D (Part-Time) Degree at Sri Vijay Vidyalaya College of Arts & Science. Salem, Tamilnadu, India. Her areas of interest include Data Mining, Big Data and Wireless Networks.

Dr.D.Maruthanayagam received his **Ph.D** Degree from Manonmaniam Sundaranar University, Tirunelveli in the year 2014. He received his **M.Phil** Degree from Bharathidasan University, Trichy in the year 2005. He received his **M.C.A** Degree from Madras University, Chennai in the year 2000. He is working as **Head and Professor**, PG and Research Department of Computer Science, Sri Vijay Vidyalaya College of Arts & Science, Dharmapuri, Tamilnadu, India. He has above **19** years of experience in academic field. He has published **5** books, more than **45** papers in International Journals and **30** papers in National & International Conferences so far. His areas of interest include Computer Networks, Grid Computing, Cloud Computing and Mobile Computing.