Semantic Web 1 (2019) 1–49
IOS Press

Storage, Partitioning, Indexing and Retrieval in Big RDF Frameworks: A Survey

Tanvi Chawla [a,*,**], Girdhari Singh [a], E.S. Pilli [a] and M.C. Govil [b]
[a] *Department of Computer Science and Engineering, Malaviya National Institute of Technology, Jaipur, India*
E-mails: 2015RCP9023@mnit.ac.in, gsingh.cse@mnit.ac.in, espilli.cse@mnit.ac.in
[b] *Director, National Institute of Technology, Sikkim, India*
E-mail: director@nitsikkim.ac.in

Editors: First Editor, University or Company name, Country; Second Editor, University or Company name, Country
Solicited reviews: First Solicited Reviewer, University or Company name, Country; Second Solicited Reviewer, University or Company name, Country
Open reviews: First Open Reviewer, University or Company name, Country; Second Open Reviewer, University or Company name, Country

Abstract. Resource Description Framework (RDF) is increasingly being used to model data on the web. RDF model was designed to support easy representation and exchange of information on the web. RDF is queried using SPARQL, a standard query language recommended by W3C. The growth in acceptance of RDF format can be attributed to its flexible and reusable nature. The size of RDF data is steadily increasing as many government organizations and companies are using RDF for data representation and exchange. This resulted in the need for developing distributed RDF frameworks that can efficiently manage RDF data on large scale i.e. Big RDF data. These scalable distributed RDF data management systems competent enough to handle Big RDF data can also be termed as Big RDF frameworks. The proliferation of RDF data has made RDF data management a difficult task. In this survey, we provide an extensive literature on Big RDF frameworks from the aspect of storage, partitioning, indexing, query optimization and processing. A taxonomy of the tools and technologies used for storage and retrieval of Big RDF data in these systems has been presented. The research challenges identified during the study of these systems are elaborated to suggest promising directions for future research.

Keywords: Storage, partitioning, indexing, query optimization, semantic web

1. Introduction

RDF data management is the core of Semantic Web and RDF stores form an indispensable part of that. Some of the centralized systems that have been introduced in the past years include RDF-3x [78], SW-store [1], Jena [73], Sesame [16], Hexastore [106] etc. Such single machine systems suffer from the bottleneck of performance for both loading and querying of large scale RDF data [21]. The increased use of RDF format for publishing data on the web has resulted in some pressing issues such as system performance and scalability. The rapidly increasing RDF data size has led to high computational and storage requirements and has stressed the limits of single machine processing [89]. The main advantage of these systems is that they incur no communication overhead however, their performance is limited by the memory capacity and computational power of a single machine [109]. Some other centralized RDF systems include Virtuoso, 3store[45], Rstar [68], DB2RDF [15], gStore [115], chameleon-db, TripleBit [111] and BitMat [12].

*Corresponding author. E-mail: 2015RCP9023@mnit.ac.in.
**Do not use capitals for the author's surname.

Some of the challenges associated with Big RDF processing are distributed storage and query processing. These challenges present the need for Big RDF management systems that can efficiently handle large volume of RDF data. Thus, for tackling this challenge of Big data the research progressed towards clustered RDF stores [87]. In contrary, to the centralized RDF systems these distributed systems have higher aggregate memory sizes and processing capacity. The first attempt in this direction was 4Store [46], in this framework the cluster is divided into storage and processing nodes. Some other clustered stores that make use of a specialized cluster are Clustered TDB [82], Virtuoso Cluster Edition [31], YARS2 [47] etc. In these systems, the data is maintained on multiple nodes. The disadvantage of using a specialized cluster is that it requires a dedicated infrastructure [97]. Also, the operations of these systems is more similar to the centralized RDF stores and they offer lower loading speed [21].

For tackling the issue of Big RDF data management many Big RDF systems use Hadoop MapReduce and other cloud-based frameworks for coordinating query processing across a cluster of nodes. These frameworks include Huang et al. [52] which processes queries locally and uses MapReduce for joining the intermediate results. The frameworks like Partout [34], DREAM [42] and JARS [92] do not use cloud computing technologies and instead have a centralized RDF store like RDF-3x installed at each slave node to process queries locally. The intermediate results are joined at a slave node and returned to the master. The frameworks like Trinity.RDF [112], TriAD [41], AdHash [43], Rainbow [39], ScalaRDF [51] are based on an in-memory engine.

RDF data model is a concise and flexible model that is suitable for representing metadata of resources on the web [69]. Thus, for representation of structured and unstructured data RDF is becoming a de facto standard. This model eases the representation and exchange of information on the web. The popularity of this model can be attributed to its fundamental graph-based model thus, any type of data can be easily expressed in this format [32]. One of the application areas of RDF model that requires modeling of large scale or Big RDF data is social semantic domain. Many government organizations and large companies have begun to use RDF as a data representation format for better product search, search engine optimization etc. The Linked Open data (LOD) cloud is a web of data that contains data from a diverse set of domains such as geographic locations, films, scientific data etc., here the data is linked to form a large RDF data cloud [69]. Thus, the rapid increase of RDF data is an outcome of this linked open data movement as enormous amount of data on the web is being represented using RDF.

In this paper, we focus on the Big RDF frameworks. We discuss the frameworks based on; (i) cloud computing technologies such as SHARD [93], H2RDF+ [85], S2RDF [99] etc.; (ii) in-memory computation engines such as AdHash [43], Trinity.RDF [112] and TriAD [41]; (iii) centralized RDF stores such as DREAM [42], Partout [34], DiploCloud [108] etc. The target of this paper is threefold. The first target is to provide an overview of the different storage strategies used in Big RDF frameworks. The second is to classify and analyze the partitioning and indexing approaches used in these frameworks to improve query performance. The third is to discuss and identify the popular retrieval mechanisms used in these frameworks. It should be stressed that the number of Big RDF frameworks discussed in this paper is not exhaustive as including all of them was not feasible.

Our contributions in this paper can be summarized as follows:

- We provide an in-depth classification of the storage, partitioning, indexing, query optimization and processing strategies used in Big RDF frameworks.
- We discuss a generic Big RDF framework and some other generic frameworks after examining the discussed approaches.
- We identify the commonly used storage, partitioning, indexing and retrieval strategies used in these frameworks.
- A comparative analysis of storage and retrieval in the discussed frameworks is presented.
- The tools and technologies used for storage and retrieval used by the frameworks are categorized.
- Finally, we identify some research challenges in Big RDF frameworks.

The proliferation of RDF data and its wide adoption as a data storage model calls for scalable Big RDF management frameworks. To tackle this challenge many Big RDF frameworks have been proposed. A comprehensive survey of the various approaches for storage, partitioning, indexing, query optimization and finally, query processing is provided in this article. We organized the rest of this paper as follows. In Section II, we discuss

some related surveys on Big RDF data and; also provide, some background information on Big RDF storage and retrieval. We introduce the storage in Big RDF frameworks according to the technique used for storage in Section III. The preparation i.e. partitioning and indexing of Big RDF data is covered in Section IV. The retrieval of Big RDF data in Big RDF frameworks is discussed in Section V according to the querying mechanism. We discuss a number of research challenges in Section VI and in Section VII we finally conclude the article.

2. Background

The proliferation of RDF data has been a pressing issue and a motivating factor for developing scalable solutions for RDF data management that are capable of efficiently storing, partitioning, indexing and querying RDF data [67]. In this section, we discuss the related surveys on RDF data storage and retrieval. The existing storage schemes, partitioning and indexing techniques and; the SPARQL query optimization and processing strategies for Big RDF data are also discussed in this section.

2.1. Related surveys

Some survey papers that discuss the RDF data management and relevant frameworks are Faye et al. [32], Kaoudi et al. [58], Özsu [83], Ma et al. [69], Abdelaziz et al. [3], Pan et al. [84], Elzein et al. [30] and Wylot et al. [109].

Faye et al. [32] discuss the different RDF storage approaches i.e. Triple Table, Property Table and Binary Table. The authors also examine the indexing approaches used by different RDF frameworks i.e. Yars, Kowari, Virtuoso, RDF-3x, Hexastore, BitMat etc. This paper focuses on the storage and indexing in Centralized RDF frameworks. The authors recognized that these centralized storage frameworks have limitations both in terms of fault tolerance and scalability. They state that no single approach is suitable, but it is a combination of different concepts that form state-of-the-art in RDF data management. The authors do not discuss the distributed RDF frameworks and concentrate only on centralized RDF frameworks.

Özsu [83] discussed different centralized approaches for RDF data management and processing. In distributed RDF management four classes of approaches are examined i.e. partitioning-based approaches, cloud-based solutions, partial evaluation-based approach and federated SPARQL evaluation systems. The storage approach used in some cloud-based distributed frameworks such as SHARD [93], H2RDF [86] etc.; partitioning-based distributed fra-meworks like Huang et al. [52], WARP [50], TriAD [41]; federated distributed frameworks i.e. DARQ, SPLENDID etc. and partial query evaluation based distributed frameworks like gStore etc. has been evaluated. The authors do not give a detailed comparison of these frameworks and also distributed query processing in these systems has not been discussed. Özsu highlighted only some main approaches .

Ma et al. [69] focus on relational database and NoSQL database based techniques for Big RDF storage. The RDF data stores are only concentrated upon in this paper but the indexing and querying techniques used in these stores in spite of being closely relevant are not discussed in depth. The current RDF stores are broadly classified into centralized and distributed RDF stores. The NoSQL database stores come under the category of distributed RDF stores. Some centralized stores that are discussed are Hexastore, RDF-3x etc. The distributed NoSQL-based RDF stores discussed are Sun et al. [102], Franke et al. [33] etc. Apart from these some other distributed stores discussed are Husain et al. [54], SHAPE [62] etc. The authors also discuss the benchmark RDF datasets like LUBM, SP2Bench etc. This survey, doesn't discuss many distributed RDF stores, no details about indexing and querying in distributed RDF stores are given also, the authors do not compare these stores.

Abdelaziz et al. [3] focus on the distributed RDF systems and thus, perform an extensive experimental evaluation of 12 of these systems i.e. SHARD [93], SHAPE [62] etc. The metrics used to analyze these systems are start-up cost, query efficiency, scalability and adaptability. These systems are categorized on the basis of their execution model i.e. Graph-based and MapReduce, secondly, Specialized RDF systems. The specialized RDF systems are further categorized according to the used partitioning scheme i.e. sophisticated, lightweight and graph partitioning. The authors concentrate on experimental evaluation of distributed RDF systems and on identifying their strengths and weaknesses. The storage, partitioning, indexing and retrieval in these frameworks is not discussed in detail.

Parameters	Kaoudi et al. (2015) [58]	Wylot et al. (2018) [109]	Our Survey
Frameworks classification	Technique-based	Tool-based	Technique-based
Cloud-based Big RDF frameworks	MapReduce-based, NoSQL database-based	MapReduce-based, NoSQL database-based, Spark-based	MapReduce-based, NoSQL database-based, Spark-based
Specialized Clustered Big RDF frameworks	NA	Considered	Not Considered
Partitioning strategies classification	NA	NA	Considered
Indexing schemes classification	NA	NA	Considered
Query processing approaches classification	Considered	NA	Considered
Generic frameworks	Not presented	Not Presented	Presented
Performance evaluation metrics	Not presented	Not presented	Presented

Table 1
Summary of related surveys

Pan et al.[84] discussed only some distributed RDF systems i.e. Huang et al. [52], H2RDF+ [85] etc. The authors also gave an experimental comparison of RDF-3x, Husain et al. [53] and H2RDF [86]. The distributed frameworks they have reviewed are only limited as well as are not discussed in depth while we examine and compare a number of distributed RDF frameworks from all crucial aspects i.e. of storage, partitioning, indexing and querying. The limitations of these frameworks are not identified and the comparison is restricted to only a few discussed distributed frameworks.

Elzein et al. [30] reviewed some Semantic Web repositories like RDF-3x, BigOWLIM, Jena and Sesame. The authors presented a taxonomy of the RDF data storage and retrieval solutions and its challenges. Some distributed RDF systems based on key value stores like Rya [91], CumulusRDF [61], AMADA [9], Stratustore [100], H2RDF [86] and MAPSIN [96]. have also been reviewed. The authors do not discuss these systems in depth and do not highlight their limitations. These frameworks are only briefly discussed and the important aspects like partitioning and indexing are not closely evaluated. Also, the authors only discuss the Big RDF frameworks using MapReduce processing framework and do not examine the other relevant systems based on the Spark framework.

Kaoudi et al. [58] provide an exhaustive survey of Big RDF frameworks and classified them based on their central functionality of storage, query processing and reasoning. The benefits of each of these frameworks are also highlighted. Some of the distributed RDF frameworks examined in depth are SHARD [93], PigSPARQL [97], etc. This paper is on the similar lines as our survey however, the authors do not discuss the Big RDF systems based on the Spark framework. In contrary, we examine the Big RDF systems using Spark framework for storage or query processing or both. We also identify the limitations of most of the discussed Big RDF systems. However, we have not taken into consideration the reasoning aspect as is taken by Kaoudi et al. We instead only concentrate on partitioning, indexing and query optimization.

Wylot et al. [109] discussed centralized like Jena, Hexastore, RDF-3x, gStore [115], Bitmat, TripleBit [111] etc. and distributed like SHARD [93], Jena-HBase [60], S2RDF [99], Partout [34] etc. RDF management systems. In contrary, we concentrate only on Big RDF frameworks. The authors classified these frameworks based on the tools they use like Hadoop, NoSQL databases, Spark etc. We have classified these systems based on four central aspects of scalable RDF management i.e. storage, partitioning, indexing and query processing. Wylot et al. do not examine these aspects and do not highlight the flaws in these systems. The authors have only broadly classified these systems while we present an in-depth comparison.

Table 1 summarizes how our survey adds in to the existing literature surveys available on RDF management.

2.2. Big RDF Storage

RDF storage is one of the primary concerns while managing RDF data and as this data continues to scale RDF data management becomes more complicated. We term RDF data as Big RDF owing to the fact that RDF datasets have a characteristic in common with Big data i.e. “Volume". Figure 5 shows a classification of the different RDF storage models. The storage schemas for managing Big RDF data are discussed below.

2.2.1. Triple Table

The first approach for RDF storage is known as vertical storage or triple table. In this format, the RDF data is stored in three columns (subject, predicate and object). The architectures which use a triple-table approach are Oracle, Sesame [16], 3-Store [45] etc. This approach is simple as all the RDF triples need to be stored in a single table in their respective columns. The drawbacks with this storage method is that as the data size increases the size of table also grows making it difficult to store all data on a single system. So, this is not the most efficient way for storing Big RDF data. This approach may be efficient for storing small RDF datasets on centralized RDF systems. But for large scale RDF data that needs to be stored on a distributed system this is not an efficient approach. Secondly, this method requires a large number of self-joins for query execution which is an expensive operation as it will require the triple-table to be joined against itself. An example of a simple Triple Table is shown in Figure 1.

Subject	Predicate	Object
Anjali Gupta	LivesIn	"Mumbai"
Kiran Sharma	WorksIn	"TCS"
Ashish Pandey	WorksIn	"HCL"
Kriti Mehra	LivesIn	"Delhi"
Anjali Gupta	WorksIn	"Infosys"
Kiran Sharma	LivesIn	"Delhi"
Kriti Mehra	WorksIn	"Capgemini"
Anjali Gupta	WorkProfile	"Testing"
Kiran Sharma	WorkProfile	"Development"

Fig. 1. Triple Table

2.2.2. Property Table

The second approach, is the property table or n-ary table. Here, the triples are stored in wide horizontal tables with n-ary columns. The idea is that seperate n-ary tables are created for subjects that have common properties. Some of the centralized frameworks which use the property-table approach are Jena [73], DB2RDF [15] and 4store [46]. This storage schema can be classified into two types: Clustered property table and Property class table. The Clustered property table stores a group of properties that tend to be defined together. The ‘type’ property is exploited by the property-class table which groups subjects associated with similar type into the same table. This approach is very efficient for star pattern SPARQL queries as it diminishes the subject-subject joins in these queries. But this approach also has some serious limitations such as a property table may contain significant number of null values which makes the property table sparse and introduces the additional overhead of storage and querying of these null values. Another limitation, is the difficulty in representation of multi-valued attributes in a property table. A sample Property Table is depicted in Figure 2.

Subject	LivesIn	WorksIn	WorkProfile
Anjali Gupta	"Mumbai"	"Infosys"	"Testing"
Kiran Sharma	"Delhi"	"TCS"	"Development"
Ashish Pandey	NULL	"HCL"	NULL
Kriti Mehra	Delhi"	"Capgemini"	NULL

Fig. 2. Property Table

2.2.3. Binary Table

The third approach, which is much more efficient for Big RDF data and has been used in many scalable distributed RDF systems is Vertical Partitioning or Binary table or Decomposed Storage Model (DSM). An example of a Binary Table is illustrated in Figure 3. A single two-column table is defined for each property of the RDF dataset in this schema. Suppose, if there are n distinct properties in the RDF dataset it means, there are n binary tables. Each binary table contains two columns (subject and object) for each property. The binary table approach overcomes the limitations of null values (as it simply eliminates the subject-values which do not have a defined property) and multi-valued properties associated with the property table approach. This storage scheme has the limitation that if a SPARQL query requires to query multiple properties then multiple binary

tables need to be accessed and joined. The second limitation is that the insert operations will be slow as multiple tables in different partitions may need to be accessed for updating value of the same subject.

LivesIn

Subject	Object
Anjali Gupta	"Mumbai"
Kiran Sharma	Delhi"
Kriti Mehra	"Delhi"

WorksIn

Subject	Object
Anjali Gupta	"Infosys"
Kiran Sharma	"TCS"
Ashish Pandey	"HCL"
Kriti Mehra	"Capgemini"

WorkProfile

Subject	Object
Anjali Gupta	"Testing"
Kiran Sharma	"Development"

Fig. 3. Binary Table

2.2.4. Mixed (Property+Binary Table)

The fourth approach, is termed as mixed as it tries to overcome the limitations associated with the above three approaches. This scheme is either data centric or workload centric as it clusters similar RDF data based on structure of RDF data or information collected from query workload. This approach uses a combination of property (or n-ary table) and binary (vertical partitioned or DSM) tables. Here, prior to RDF partitioning and data distribution in a distributed environment, the related RDF properties are grouped together using clustering. This group of related properties are collected by scanning the input RDF data or query workload. The objective is to determine the set of properties which exist together for a large number of subjects in the RDF data or determine the number of queries where these properties are frequently queried together. This scheme is a balanced mix of both the approaches. Unlike, the property table approach this scheme doesn't suffer from the problem of null values and multi-valued attributes. Also, this scheme reduces joins by trying to maintain as much RDF data that is given as input or as much RDF data that is queried together in the same table.

2.2.5. Graph-based

The RDF data is represented using nodes and edges as a directed labeled in the graph-based RDF storage model. For eg., the triple (s hasProperty o) can be interpreted as an edge labeled with hasProperty from node s to node o. The systems that are built on the graph processing API or graph databases generally use this model. With the graph model, RDF data can be visualized conveniently. The RDF data is modeled in its native graph form in Trinity.RDF [112]. In the graph form, Trinity.RDF models RDF data where entities (subjects; s and objects; o in RDF triples) represent the nodes in graph and graph edges represent the relationships (predicates; p of RDF triples). Here, each graph node with a unique id represents an RDF entity and is stored as a key-value pair in Trinity. The graph-based model has some advantages over the triple-based model that new data sources can be easily added and linked also, it is more flexible.

2.3. Big RDF Partitioning

RDF Partitioning is the second issue which is faced while handling Big RDF data. The basic idea of this solution is to place the tuples that may be involved in the join into the same node. The particular partitioning algorithm can make a large difference in the amount of data that needs to be shipped over the network at query time when data is partitioned across multiple machines [52]. A good partitioning solution can aid in cutting down the cost of query evaluation by significantly scaling down or absolutely avoiding the cross-node joins [105].

2.3.1. Hash Partitioning

Some of the most prominent partitioning approaches are hash, list, range partitioning etc. In the simplest form of hash-based data partitioning a hash function is applied for distributing RDF data in a cluster on a component of the RDF triple. In the hash partitioning technique, the RDF triples are equally distributed among specified number of partitions based on the partition column. The hash partitioning can be done on RDF data by hashing on any RDF element i.e. subject, predicate or object. In [2], the authors observed the user query pattern to identify the partitioning element and it was observed that partitioning on the predicate column is more efficient and more effective results are obtained in terms of query processing. Some of the systems which partition RDF data by hashing are; SHARD [93], Virtuoso Cluster, Clustered TDB [82], 4store [46] and Yars2 [47]. But, the triples are commonly hashed by their subject such as in AdPart [44] for guaranteeing that star queries (i.e. queries that contain only subject-subject joins) can be locally evaluated. The drawback with hash partitioning is that it doesn't take the graph structure of RDF data into account and thus, the nodes that are near to each other in a RDF data graph may not be stored on the same machine resulting in considerable distributed query processing. It is observed that hashing is a unsuitable partitioning strategy for such applications that have diverse query workloads.

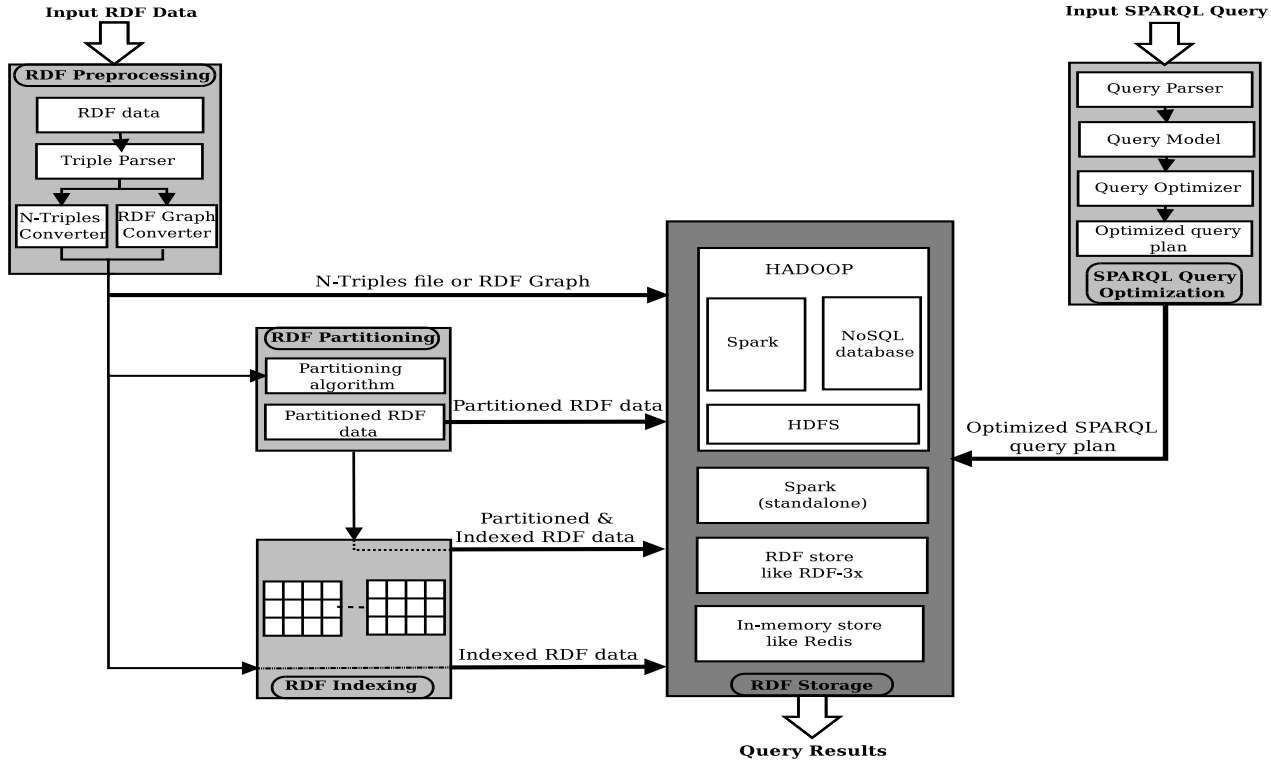

Fig. 4. A generic Big RDF Framework

2.3.2. *Graph Partitioning*

In graph partitioning, the triples that are adjacent to each other in a RDF data graph are stored on the same machine. Thus, this approach overcomes the randomness that can be an issue in the purely hash-based partitioning techniques as used in some systems such as SHARD [93]. One of the most popular Big frameworks that uses graph partitioning is proposed by Huang et al. [52]. Graph partitioning increases the chances of placing the nodes that are highly interconnected on the same machine thus; in turn increasing the probability of query answers being computed locally. In graph-based partitioning, generally a min-cut graph partitioning software such as METIS is used that takes a RDF graph G and the partition size n as input and it outputs the n disjoint sets of nodes of G. All these sets are of similar sizes and the number of edges in graph connecting nodes in the distinct sets is minimised [90]. The drawback with this scheme is that it doesn't provide guarantee that common queries like star queries can be locally evaluated. Thus, significant amount of distributed computation may be required for guaranteeing completeness even in such cases where a query can be fully answered by the locally computed answers.

2.3.3. *Workload-aware Partitioning*

The workload-aware partitioning strategies distribute RDF data based on historic query workload. Some of these strategies work on the assumption that there is not much change in query workload over a period of time. Thus, they learn from the historic workload the frequently queried triples. They use this knowledge for partitioning the RDF data optimally for future SPARQL queries. Some of the frameworks which consider the query workload for partitioning process are; Partout [34], AdHash [43], WARP [50], Diplocloud [108] etc. One of such systems which analyzes queries contained in the historic query workload is Partout. WARP takes into account the query workload for choosing such parts of RDF data that need to be replicated. It does not replicate the RDF data that is rarely used thereby, diminishing the storage overhead. Diplocloud follows an adaptive partitioning strategy and it adapts to the dynamic workload automatically. AdHash and DiploCloud overcome the drawbacks of Partout and WARP which as-

sume a static workload for partitioning RDF data and thus, do not adapt to the changes in workload.

2.3.4. Horizontal Partitioning

Horizontal (or row-wise) partitioning may also take the query workload into account to partition RDF data. The workload can be considered for determining optimal partitions to apply horizontal fragmentation of relations i.e. partitioning data in accordance to the predicates present in frequent queries. Thus, it becomes possible to fully evaluate these queries on a single partition. Some of the systems which use this technique are; Partout [34], TriAD [41] etc. Partout performs horizontal fragmentation by considering the query workload. The horizontal partitioning can be used in combination with hash partitioning like in TriAD. In horizontal hash partitioning, the RDF data is generally partitioned into files by subject. Thus, triples that have a common hash value of subject are partitioned into same file. The horizontal hash partitioning will not be efficient for retrieval of suitable triples when the subject is a variable in the queries [80]. In contrast, to vertical partitioning this kind of partitioning complements to the case where all predicates can be found in several data sources [70].

2.4. Big RDF Indexing

The primary objective of indexing is to make search and access of Big RDF data easy at any given time. The efficiency of RDF data retrieval is dependent upon its physical organization and indexing. The search on RDF indexes is done by using the basic graph patterns (BGP). Abburu et al. [2] propose a predicate-centric partitioning and multiple indexing approach. The indexes built in this approach are P, PS, PO, PSO, POS. The authors apply this multiple indexing strategy on all the different combinations of predicate centric RDF elements.

2.5. SPARQL Query Optimization & Processing

There is a considerable amount of work available on SPARQL query optimization used by centralized and distributed RDF systems. Many of these works use the traditional optimization techniques for example, selectivity-based join reordering, pushing projections etc. One of the popular approaches for SPARQL query optimization is by reordering the triple patterns for minimizing the cardinality of intermediate results [74]. The triple patterns in a SPARQL query are similar to RDF triples such that each of the RDF triple elements i.e. subject, predicate and object may be a variable or a literal [56]. The Big RDF frameworks like PigSPARQL [97] use the Variable Counting approach proposed by Stocker et al. [101] for optimization of SPARQL queries by reordering the triple patterns in a query. The most popular techniques used by RDF systems for SPARQL query processing is subgraph matching. Some of the systems which use subgraph matching are gStore [115], Bahrami et al. [13], Kassaie et al. [59] etc.

We depict a generic Big RDF Framework in Figure 4. As shown in figure, a Big RDF framework comprises of five components i.e. Storage, Partitioning, Indexing and SPARQL Query Optimization and Processing. These components together form a Big RDF framework. The Big RDF Preprocessing module pre-processes the RDF data to convert it into a suitable form. The pre-processed data in N-Triples format or in native graph form may be directly stored in the storage module or the data may be fed as input to Big RDF Partitioning (2^{nd} module) or Big RDF Indexing (3^{rd} module) modules. There are 4 cases for transforming Big RDF data i.e. No Partitioning & No Indexing, Partitioning & No Indexing, No Partitioning & Indexing and Partitioning & Indexing. The Big RDF data is transformed into either of these 4 forms and is stored in the Big RDF storage module.

The Big RDF storage module comprises of the Hadoop framework, the centralized RDF stores and the in-memory stores. The MapReduce or the Spark processing frameworks of Hadoop may be used for query processing and the transformed data may be stored in HDFS or a NoSQL database like HBase. Some of the Big RDF frameworks store this transformed Big RDF data in HDFS and push the query processing to centralized RDF stores like RDF-3x. While, others store this data in centralized RDF stores and push the query processing to MapReduce. The 5^{th} module is responsible for optimization of SPARQL queries for improving the query performance. The optimized query plan obtained as output from this module is executed on the Big RDF data stored in the Storage module.

3. Big RDF Storage

A classification of schemes for storage of Big RDF data has been provided in this section as illustrated in Figure 5. Each of these schemes has been analyzed from

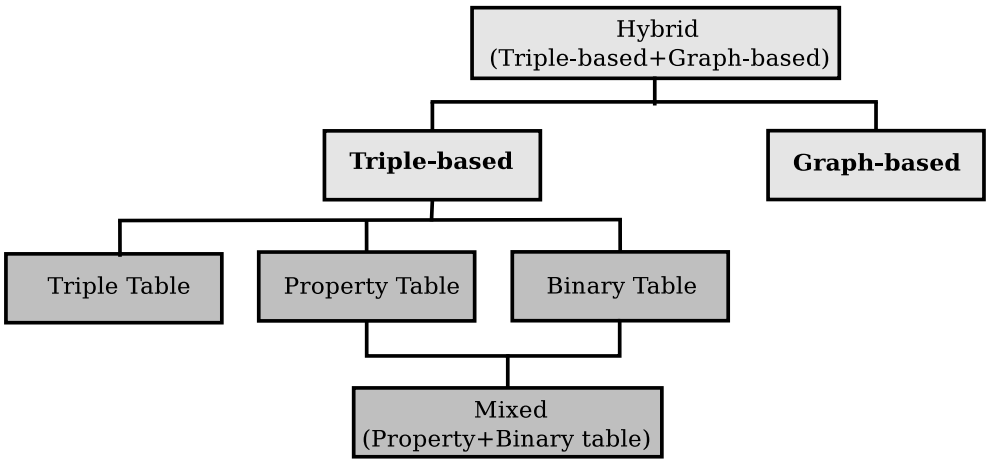

Fig. 5. A classification of Resource Description Framework (RDF) storage models

the perspective of scalable Big RDF storage as storage has a great impact on query evaluation. In this section, we classify the Big RDF frameworks according to the used storage schema.

In PrestoRDF [72], the four different RDF storage schemes are implemented by RDF-loader which also creates external Hive tables for storing metadata in the Hive Thrift server. The four different storage strategies implemented are: Triple Table, Property Table (or wide table), Horizontal store and Vertical Partitioning (or binary table). The authors compare the three storage schemes i.e. triple store, vertical store and horizontal store.

3.1. Triple Table

In this section, we discuss the Big RDF frameworks that use a Triple Table storage format for Big RDF data. Further, we sub-divide these frameworks on the basis of how and where Big RDF data is stored in a three column triple table format.

Khadilkar et al. [60] proposed Jena-HBase, a distributed framework that supports scalable storage and querying for Big RDF data. This framework supports multiple storage layouts with HBase i.e. Vertical Partitioned, Indexed, Vertical Partitioned and Indexed, Hybrid and Hash. With these custom storage layouts Jena-HBase provides tradeoff in terms of data storage and query performance. From results, it is observed that Hybrid layout is superior than others in performance as it combines the advantage of simple and vertical partitioned layouts. The architecture of Jena-HBase comprises of several HBase tables with different layouts to store Big RDF data.

Mammo et al. [72] proposed an architecture PrestoRDF based on Presto for processing Big RDF data. Here, data is stored in HDFS and an in-memory processing engine is used to process it instead of MapReduce. The key component of this framework is RDFLoader that is used for reading, loading and parsing RDF triples. Some of the other components of PrestoRDF are a command line interface (CLI), Facebook Presto, Hive Metastore, a SPARQL to SQL compiler (RQ2SQL) and HDFS.

3.1.1. Index Tables

Sun et al. [102] proposed a scalable Big RDF store where the RDF triples are stored in six HBase tables. These tables cover all permutations of the RDF triple patterns. In this framework, the authors adopted the Hexastore scheme for RDF data storage in HBase. Here, all the possible ordering of RDF triples are indexed for storing data in HBase tables. Thus, the resultant six tables in HBase allows retrieval of any possible triple pattern in a SPARQL query with a single lookup on one of the tables (except for the triple pattern with three variables).

Stein et al. [100] proposed Stratustore that uses AmazonâĂŹs SimpleDB as back end to store Big RDF data using entity oriented mapping, this system extends the Jena Semantic Web framework. The evaluation of Stratustore is done using Elastic Compute Cloud (EC2), Amazon Simple Storage Service (S3) and SimpleDB. The RDF data model is mapped to SimpleDB by using entity oriented mapping. A large proportion of

SPARQL queries can be pushed into SimpleDB by using this entity oriented mapping. As SimpleDB doesn't support bulk loading so the interface of SimpleDB was changed to allow bulk loading. Stratustore groups the triple patterns by subject.

Ladwig et al. [61] proposed CumulusRDF system that uses one of the key value stores i.e. Apache Cassandra as its storage backend. The linked data lookups and basic triple pattern lookups are supported by this framework. In CumulusRDF, the authors compared two indexing schemes for the RDF triples on top of key value stores. The authors implemented two storage models i.e. standard key-value storage model or the flat layout and hierarchical layout which is based on supercolumns. The features provided by data model of Cassandra that are used by CumulusRDF are; supercolumns (which act as a layer between the row and column keys) and the secondary indices (to provide value key mappings for columns). This framework doesn't use any dictionaries for mapping the RDF terms and the original data is instead stored as column keys and values.

Franke et al. [33] proposed schemes for distributed storage and querying in HBase and MySQL clusters. The proposed framework is based on the cloud computing and relational database technologies, and the authors compare these two schemes. A novel database schema is presented for storage of RDF data in HBase. Some efficient algorithms are also implemented in this framework for triple and basic graph pattern (BGP) matching in HBase according to proposed storage schema. An efficient SPARQL to SQL translation algorithm is also proposed. The authors compare these two cluster approaches i.e. HBase and MySQL for storage and querying of Big RDF data.

Papailiou et al. [86] proposed H2RDF, a framework that selects a join scenario which is most advantageous for any given join by providing an adaptive choice to choose between the centralized and full distributed modes of MapReduce-based query execution. The simple and multi-join queries are supported by this framework. H2RDF stores Big RDF data in HBase and bulk import MapReduce jobs are used for loading, indexing and maintaining statistics on Big RDF data. The architecture of H2RDF, comprises of Jena SPARQL parser (for ensuring correctness of syntax and creating a query graph), Join planner (for iterating over the query graph and greedily choosing the join to be executed) and Join executor (decides which algorithm to use for every join).

Schätzle et al. [96] proposed MAPSIN (Map-Side Index Nested Loop Join) which uses the indexing properties of distributed NoSQL store HBase for improving the performance of selective SPARQL queries. HBase is used in this framework as the storage layer for storing Big RDF data instead of HDFS. Thus, this avoids the shuffling of join patterns across the network and only relevant join partitions are accessed in each iteration. These MAPSIN joins are completely processed in the map phase. Also, no additional shuffle and reduce phase are required by MAPSIN to preprocess the data for consecutive joins. This algorithm is optimized for efficiently processing multi-way joins. MAPSIN uses the Map-Reduce framework for processing of SPARQL BGP's. The storage schema in this framework is inspired by Franke et al. [33] and thus uses only two tables.

Papailiou et al. [85][87] propose H2RDF+ that adapts between single and multi-machine query execution based on the complexity of query join. Here, bulk import jobs are used for loading and indexing Big RDF data. Also, aggressive compression is applied to optimize distributed index and thus minimize required storage space. This system implements scalable and distributed MapReduce-based versions of multi-way merge and sort-merge join algorithms.

Choi et al. [22] proposed RDFChain that supports storage and retrieval of Big RDF data by using a combination of MapReduce and HBase. A cost-based map side join is used in this framework for reducing the number of map jobs since, a single map job using some statistics is required here for processing as many as possible joins. The authors overcome the drawbacks of MAPSIN algorithm [96], that is optimized for star pattern queries but not for the chain pattern queries. RDFChain provides an improvement over MAPSIN by storing RDF data having subjects that appear as objects in a separate T_{com} table. Thus, this strategy gives an optimal performance for chain pattern queries. The proposed chain centric storage considers every possible join pattern in its storage schema and thus reduces the number of storage access.

Li et al. [66] proposed SHOE (SPARQL on Hadoop with Optimization Encoding) a MapReduce based SPARQL query processing engine for handling Big RDF data. The proposed framework consists of three major components i.e. RDF data loader, partition gener-

ator and query processor. Here, an integer identification is built for each item by translating the triples into a set of identification based counterparts thereby, reducing the volume of RDF data. Then, the SPARQL query is carried out by optimizing execution server according to the partitions that are derived from the permutations and combinations of subject, predicate and object (s,p,o).

Gu et al. [39] proposed Rainbow, a distributed and hierarchical Big RDF framework with dynamic scalability. The HBase cluster is taken here as the persistent storage and Redis cluster as second storage. In case of failure, like in a scenario where the Redis node gets failed Rainbow switches automatically to the HBase storage layer for querying. Rainbow comprises of three layers; bottom for distributed persistent storage i.e. HBase, middle for distributed memory storage that consists of single node in-memory stores i.e. Redis and top layer for access that provides data loading and querying to users. In Rainbow, the elements of a RDF triple i.e. subject, predicate and object are encoded and stored to improve the space utilization of disk and memory storage also, to reduce the intermediate data size.

Xu et al. [110] proposed SparkRDF, an in-memory Big RDF framework. SparkRDF manages open social network data (OSN) using a key value store i.e. HBase for RDF storage and an in-memory pipelined strategy for query processing. It is a hybrid two-layer RDF management framework designed for supporting efficient querying over Big RDF data. The first layer in SparkRDF, consists of an indexing schema with three tables which is implemented in HBase. This schema supports import and indexing of Big RDF data. In the second layer, iterative joins are processed using Spark RDD for SPARQL processing. The bulk loading process of MapReduce is used to handle Big RDF data, as it minimizes network and I/O operations.

Punnoose et al. [91] proposed Rya, a scalable Big RDF framework to store and retrieve Big RDF data in a distributed cluster. A new serialization format is used for Big RDF storage, the indexing methods used here support fast data access and the implemented query processing techniques speed up SPARQL query evaluation. All key value pairs are sorted and partitioned on row ID part of the key in Accumulo. The grouping and sorting of rows is done on the basis of lexicographical sorting of the row ID. As a result, the rows with similar IDâĂŹs are grouped into same machine to support efficient and fast access. The row ID part of Accumulo tables stores RDF triples. The timestamp associated with each RDF triple (i.e. by default its insertion time) is recorded with each triple into the table. Rya stores all data related to the triple in the row ID in Accumulo.

Oh et al. [81] proposed a framework that runs a job optimized query plan to decrease the amount of intermediate data. An efficient map-side join algorithm is presented in this framework, also abstract RDF data is used here for filtering out the unnecessary RDF data. The proposed HBase schema holds all data associated with a specific join variable into a single row. Thus, this schema supports evaluation of multiple triple patterns sharing join on a variable to be evaluated in in a single MapReduce job. The architecture of this framework consists of a bulk load (comprising of a data partitioner and a triple encoder, a partition dictionary and encoded abstract RDF Data) and a query processor (query Parser, query Planner, 1st stage query executor, job planner, and 2nd stage query executor) component.

Harbi et al. [43] proposed AdHash (Adaptive Hashing) an in-memory engine that monitors the data access patterns continuously and dynamically adapts to the query workload. This is done by redistributing and replicating frequently accessed data in an incremental manner thus, reducing or eliminating the costs of communication for future queries. AdHash is based on the master-slave model, the data is partitioned by master among workers and it also collects global statistics. The functions performed by a master include receiving user queries, generating query execution plans, maintaining coordination between workers, collecting final results and finally, returning results to the user. Adhash hashes triples in each worker by the predicate.

Guo et al. [40] proposed a novel distributed SPARQL query algorithm i.e. RQCCP (RDF data Query Combined with Classes Correlations with Property) based on the Spark framework. The proposed algorithm includes two parts i.e. splitting RDF data and its storage and, data indexing and query processing. The first stage is of data parsing where the class of subject, predicate and the class of object are collected and recorded in the file. At the same time, each instance and its corresponding class is written into the instance-class mapping file. Then these two files on the Spark platform are split and uploaded after splitting to the HDFS. After the process of data splitting and storage is finished multiple data and instance class mapping files are generated.

Rajith et al. [92] proposed JARS (Join-Aware Distributed RDF Storage) a join-aware distributed RDF storage system that eliminates inter-node communica-

tion cost incurred for star pattern queries and reduces communication cost for chain pattern queries. JARS uses the PostgreSQL database for RDF storage. The authors analyzed the SPARQL queries and identified the most common type of joins involved in the query patterns i.e. Subject-Object (s-o), Subject-Subject (s-s) and Object-Object (o-o) joins. From this information the authors devised a join-aware approach where the triples having same subject/object are co-located on the same server thereby, avoiding the joins across nodes.

Hu et al. [51] proposed ScalaRDF, an in-memory Big RDF framework which provides a data store and placement protocol. ScalaRDF uses an extension of the consistent hashing protocol to place Big RDF data efficiently and achieve elasticity for scale in or out of resources. It supports easy updation of data and cluster resources. The in-memory Big RDF data is backed on the disks of neighboring (or next hop) nodes thus minimizing the impact in case of failure. To support elasticity in this system the dynamic RDF data is redistributed locally thereby avoiding the unnecessary movement of whole data. This replication feature helps in achieving a swift failover for the running queries. ScalaRDF uses the master-slave model where master parses the input files and distributes triples to the slaves. It uses Redis for in-memory storage as it is a key-value in-memory database.

Table 2, summarizes the Triple-based Big RDF frameworks that use an index table scheme for Big RDF storage.

3.1.2. Flat files

Rohloff et al. [93] proposed SHARD, a scalable Big RDF framework on top of Hadoop. SHARD persists Big RDF data in flat files in the HDFS where each line of text file represents all the triples associated with a different subject. To improve query performance of similar later queries SHARD saves the intermediate results of these queries.

Cheng et al. [20] proposed a framework for Big RDF data where data preprocessing is done using a method called PredicateLead. In the next step, the job partitioner algorithm partitions the SPARQL query into several MapReduce jobs. The output of these previous two operations is the input of last step i.e. query processing in MapReduce. The proposed framework consists of three parts i.e. data preprocessing, job partition and query execution. In this framework, the Big RDF data is stored in HDFS and Hadoop MapReduce is used for query processing. Here, the data is stored in multiple files where file name corresponds to the predicate name. In each file, a RDF triple is stored in the form of <subject, predicate#object> in a single line.

Tripathi et al. [104] proposed SARROD (SPARQL Analyzer and Reordering for Runtime Optimization on Big Data) a query reordering algorithm which reduces the response time of SPARQL queries. This algorithm leverages the fact that the order of placing query clauses or triple patterns in a query affects the query response time. The SHARD triple store was used for the execution and evaluation of SARROD and the proposed algorithm was tested on the LUBM dataset. The response time of queries was compared for the 3 different query orderings i.e. default, sorted and reverse. SARROD gives an optimal query ordering and it is seen that a speed up is observed in query processing if the queries are reordered in an optimal manner.

Aranda-Andújar et al. [9] proposed AMADA, a Big RDF management framework implemented on top of the Amazon Web Services (AWS) cloud infrastructure. After the users upload Big RDF data this framework supports storage, indexing and querying large volume of RDF data and operates as Software as a Service (SaaS) approach. This framework stores RDF data on the cloud, and the access path selection is done using an index-based mechanism that reduces the query processing time. In AMADA, the Big RDF data is stored in a distributed manner in Amazon Simple Storage Service (S3). In S3, a URI is assigned to each dataset using which it can be retrieved that can be later used while processing queries on EC2 nodes. Here, the input data is stored as a file in S3.

3.1.3. Entity-based model

Zhang et al. [113] proposed EAGRE (Entity-Aware Graph compREssion) technique where the I/O cost incurred before the map phase filtering, is reduced by using a distributed I/O scheduling solution. The distributed scheduling proposed here is based on an entity-based compression scheme for the RDF. The RDF graph is firstly transformed into an entity graph where only the nodes that have outgoing edges are kept. Here, the entities represent the subjects in RDF triples of this graph. The entities with similar properties (a similarity measure is used here to determine this) are grouped together in an entity class. The compressed RDF graph thus obtained comprises of only the entity classes and connections between them.

Author	Compared/ Tested with	Evaluation metrics	Tested dataset(s)	Weakness/Limitation(s)
Sun et al. (2010) [102]	Different SPARQL queries	Query execution time	LUBM	- The six tables need to be simultaneously accessed while modifying and deleting RDF data. - Here, HBase stores the RDF data in files on HDFS, an RDF dataset is actually stored 18 times using this schema.
Stein et al. (2010) [100]	Jena [73], Virtuoso, Sesame or Mulgara [16]	Queries per second	BSBM	- The more complex queries have a very slow runtime due to the joins that need to be executed on client side.
Ladwig et al. (2011) [61]	Two storage layouts i.e. Hierarchical & Flat	Requests per second, Query execution time	DBpedia	- CumulusRDF only supports only single triple pattern queries as it cannot handle joins.
Franke et al. (2011) [33]	Different SPARQL queries	Data ingest performance, Query execution time, Scalability	Third Provenance Challenge (PC3), LUBM	- A lot of redundancy and data hotspots issues are caused in the proposed solution and is limited in calculation of SPARQL basic graph.
Papailiou et al. (2012) [86]	RDF-3x, Husain et al. [53]	Data import time, Query execution time, Scalability	LUBM	- If a predicate is the searching key then, searching time slows because indexing predicate data (PSO or POS) are stored only in one table.
Schätzle et al. (2012) [96]	PigSPARQL [97]	Data loading time, Query execution time, Scalability	SP2Bench, LUBM	- The two HBase tables i.e. T_{spo} and T_{ops} are used here, for processing queries using Map-side joins so, during the processing of chain pattern queries, both these tables has to be accessed, thus resulting in performance decrease.
Papailiou et al. (2013) [85][87]	Husain et al. [53], H2RDF [86], RDF-3x [78]	Storage space, Data import time, Query execution time, Queries per second, Query scalability	Yago2, LUBM	- The intermediate results of iterative joins must be written back to disk in distributed mode, thereby leading to low query efficiency.
Choi et al. (2013) [22]	Husain et al. [53], MAPSIN [96]	Intermediate results size, Storage accesses	LUBM	- During processing of star pattern queries, both the T_{com} and SPO/OPS tables have to be accessed.
Gu et al. (2014) [39]	Rainbow-Hbase, SHARD [93], Jena-Hbase [60]	Query execution time, Data & Machine scalability, Dynamic scalability & fault tolerance	LUBM	- Unlike ScalaRDF [51], it doesn't support update.
Xu et al. (2015) [110]	Myung et al., H2RDF [86]	Query execution time, Query scalability	LUBM	- Due to the time for setting up a spark job, it performs a little slower in high selective queries.
Punnoose et al. (2015) [91]	SHARD [93], RDF-3x [78], Huang et al. [52]	Queries per second, Query execution time, Data loading time	LUBM	- The triple table model of Rya contains three columns, it requires to replicate the data multiple times in order to exploit the indexes over all possible elements.
Oh et al. (2015) [81]	No optimization, Partition filter, Both optimizations, RDFChain [22]	Input filtered out, Number of jobs for each query, Query execution time	LUBM	- The map-side join used here has limitations over arbitrary data sets due to preprocessing step being made a prerequisite. The drawback is that it cannot perform sort and partition according to the join key.
Harbi et al. (2015) [43]	SHARD [93], H2RDF+ [85], SHAPE [62], Trinity.RDF [112] TriAD [41]	Query execution time, Data preprocessing time	LUBM, WatDiv, Bio2RDF	- The methods used here for locality preserving are prohibitively expensive
Guo et al. (2015) [40]	Liu L et al., IMSQ	Storage space, Filtered input triples, Query execution time, Scalability	LUBM	- The intermediate results are co-partitioned on same join key so, it doesn't completely avoid shuffling.
Hu et al. (2016) [51]	Rainbow-Hbase [39], Rainbow-IM [39], gStore [115]	Scalability, Recovery (Scaleout, Fault tolerance), Average insert & delete time, Query execution time	LUBM	- Unlike Rendezvous, it doesn't maintain any cache.
Rajith et al. (2016) [92]	RDF-3x [78], SHARD [93], Huang et al. [52], SHAPE [62],	Scalability, Query response time, Data load time	LUBM	- The triples having same subject/object are stored twice so, storage space is almost doubled thus causing a huge storage overhead. - The complexity associated with update operations (as triples may be replaced on an another target node due to change in hash value.)

Table 2

Summary of Big RDF frameworks using index table storage scheme

Du et al. [28] proposed a framework where the RDF triples are logically partitioned on the basis of schemas. The query evaluation tasks are assigned to different logical partitions with these logical partitions thus, maximizing the filtering power of local nodes and reducing the communication cost. This framework uses an entity-based schema and partitions RDF triples into small partitions of entities that have the same schema.

A comparison of Triple-based Big RDF frameworks that use flat files or entity-based scheme for storage is given in Table 3.

3.1.4. Partitioned RDF data files in local storage

Huang et al. [52] proposed a scalable Big RDF management system which uses a triplestore i.e. RDF-3x across the nodes in cluster and the RDF dataset is partitioned across these data stores. The architecture of this framework is based on master-worker and at the master side the graph data partitioner is used for partitioning Big RDF data. After the RDF graph is partitioned, a triple placer at the master is used for assigning triples to the machines. Further, the communication overhead is reduced by replicating some triples on multiple machines. In this framework, a data replicator on each worker decides on-boundary triples (i.e. triples that are on the boundary of its partition) and it replicates them on the basis of some specified n-hop guarantees.

Lee et al. [62] proposed SHAPE (Semantic HAsh Partitioning-Enabled), a distributed and scalable Big RDF system which extends the simple hash partitioning method. In the proposed system, locality optimized RDF graph partitioning is combined with cost aware query partitioning for query execution over Big RDF graphs. This system is implemented on top of the Hadoop MapReduce framework where the master serves as the coordinator and the set of slaves serve as the workers. Here, the input RDF graph is loaded into HDFS. The Big RDF data is fetched into a data partitioning module that is hosted on the master machine. This data partitioning module partitions this data across slave machines. The RDF-3x triple store is installed on each slave server of the cluster and the intermediate results generated by the subqueries are joined using Hadoop MapReduce. The distributed query execution planning for each received query is performed by the master, which also serves as an interface for SPARQL queries.

Galárraga et al. [34] proposed Partout, a framework for fast processing of Big RDF data in a cluster. The architecture of Partout comprises of a central coordinator and n hosts which store the actual data in the cluster. Some of the responsibilities of a coordinator include, RDF data distribution among hosts, constructing efficient distributed query plans and to coordinate execution of SPARQL queries. The coordinator doesn't have access to actual data and it rather relies in global statistics of RDF data that are generated for query planning, at the time of partitioning. A triple store i.e. RDF-3x is run on each host node in Partout. The query workload for all queries is distributed equally among all compute nodes by this system.

Wang et al. [105] proposed a hybrid architecture for processing SPARQL queries where a RDF-query engine is used for storing Big RDF data and executing join operations while MapReduce framework is used for computation. Here, the RDF-3x query engine is installed on each worker node. The objective of this framework is to lessen the number of cross node joins to the maximum extent. After the partitions are obtained, the data loader procedure on each worker node is responsible for loading all triples in RDF-3x triple store installed at that worker machine. The query patterns are designed to accommodate as many queries as possible. The MapReduce framework is used for query execution while RDF-3x serves as the local engine.

Leng et al. [64] proposed BRGP, a balanced RDF graph partitioning algorithm for cloud storage. BRGP uses a modularity-based multi-level label propagation algorithm (MMLP) for roughly partitioning RDF graph and a balanced K-mediods clustering algorithm for final k-way partitioning. This approach uses a label update rule based on modularity for improving the coarsening quality and the efficiency of asymmetric RDF graph. The k-way partitioning is implemented here by integrating a balanced adjustment strategy based on the edge weight and vertex weight with the k-medoids. The size of original RDF graph is reduced by using an equivalent pruning strategy. The locality of information and balanced load distribution are maintained here for reducing communication overhead during SPARQL query execution and ensuring parallel execution of all storage nodes thus, fulfilling the objective of proposed partitioning strategy.

Atashkar et al. [10] proposed a partitioning strategy based on Apache Spark for Big RDF processing. The main objective of the proposed approach is to minimize communication by reducing the amount of intermediate results transferred over the network. Here, the MySQL database is used for data storage and Spark is connected

Author	Compared/ Tested with	Evaluation metrics	Tested dataset(s)	Weakness/Limitation(s)
Rohloff et al. (2010) [93]	Sesame [16]+DAMLDB, Jena [73]+DAMLDB	Query execution time	LUBM	- The RDF data is stored in plain files on HDFS resulting in scanning of entire dataset during query processing. Also, if the query contains multiple clauses the dataset may need to be scanned multiple times.
Tripathi et al. (2014) [104]	Different query orderings i.e. default, sorted & reverse	Query execution time	LUBM	- The entire dataset in triple store plus the intermediate data produced by the previous iterations needs to be accessed in each iteration that involves a MapReduce step.
Zhang et al. (2013) [113]	SHARD [93], Zhang et al. [114]	Setup time cost, Query execution time, I/O read (in MB), Network volume (in MB)	BTC 2011, Yago2, SP2Bench	- In case of longer-diameter queries or unexpected workloads the Hadoop-based joins cannot be completely avoided thus, leading to slow down in query response times by two or more orders of magnitude.
Du et al. (2013) [28]	Husain et al [53]	Entities and data distribution, Query execution time, Scalability	BTC, LUBM	- The large intermediate results incur lot of I/O & inter-node communication thereby, severely degrading the query evaluation efficiency.

Table 3
Summary of Big RDF frameworks using flat files or entity-based storage scheme

by JDBC to this database. Spark RDD is used for representing every table in the database as a seperate variable and this Spark RDD is automatically distributed in the cluster.

3.1.5. Partitioned RDF data files in Hadoop

Curé et al. [26] proposed HAQWA (Hash-based and Query Workload Aware Distributed RDF Store), a novel system implemented over the Apache Spark framework. This system proposes a trade-off between the complexity of data distribution and the efficiency of query processing. The three components in this system are; data fragmentation and allocation, data encoding and query processing. The allocation approach used in HAQWA is inspired from the WARP algorithm [50] and it is based on inspection of frequent queries executed over the dataset. The parallel execution of different tasks is ensured in this framework by implementing the component steps as Spark programs using Scala as the programming language.

Agathangelos et al. [4] proposed a distributed framework implemented over Apache Spark for classifying the streaming RDF data with the help of machine learning techniques like Logistic Regression and Random Forest (RF). This framework combines the machine learning techniques and the RDF partitioning techniques to classify the incremental data. The two major components in the proposed framework are: Data Manager and Incremental Partitioner. The metrics used for evaluating performance of the two classifiers i.e. Logistic Regression and Random Forest are Accuracy, Precision and Recall.

Naacke et al. [77][76] studied two distributed join algorithms i.e. partitioned and broadcast join for evaluating BGP expressions on Apache Spark. It is proved by the authors that the hybrid join plans have more flexibility and they perform better than the single join plans. The authors proposed a cost-based framework for distributed execution of SPARQL graph pattern queries on Spark. The authors implement the hybrid distributed join plans on different data storage layers i.e. Spark SQL, Resilient Distributed Datasets (RDD) and DataFrame (DF). The proposed SPARQL Hybrid approach combines the partitioned and broadcast join algorithms. The information about existing data partitioning is used by the optimizer for combining the joins and avoiding useless data transfer. The, a simple dynamic greedy optimization strategy based on data statistics is used for obtaining the most efficient query plan by combining the two join algorithms.

Goasdoué et al. [35] proposed CliqueSquare, a Big RDF framework based on Hadoop, MapReduce and HDFS for storage and retrieval of Big RDF datasets.

This system uses a novel RDF data partitioning technique for effectively evaluating SPARQL queries by minimizing number of MapReduce jobs and the amount of data transferred between nodes during query execution. A clique-based algorithm for query processing is proposed by authors to produce query plans having minimum amount of MapReduce stages. During query processing, the most common type of queries are executed locally at each node by using this proposed partitioning strategy. During partitioning the objective of this system is to place Big RDF data in such a manner that the maximum number of joins are evaluated in map phase itself and, such joins are called as co-located or partitioned joins.

Goasdoué et al. [36] proposed a novel optimization approach for evaluating SPARQL queries on Big RDF data in a parallel environment. This framework builds flat query plans for reducing response time of queries such that the number of joins in the root-leaf path are minimized. These algorithms used for optimization rely on star equality joins or n-ary for building flat plans. CliqueSquare focuses on logical optimization of BGP queries. These algorithms are deployed on a MapReduce-based platform. A novel generic algorithm i.e. CliqueSquare and 8 variants of this algorithm are proposed by the authors. The ability of finding the most flat plan for a SPARQL query is the basis of comparison of these variants. From comparison, it is observed that a reasonable number of flat query plans are developed by the variant CliqueSquareMSC. This variant gives a robust query performance by improving the efficiency of optimization process and making it competent for complex SPARQL queries.

3.1.6. Single table storage scheme

Chebotko et al. [18] proposed techniques for storing and indexing of RDF data in HBase that is more suitable for provenance datasets that can be stored as RDF graphs and queried using SPARQL. This work is different from the other works as it handles very large number of comparatively small RDF graphs while the existing works only manage Big RDF graphs that involve data partitioning. Here, the RDF graph is stored as one value rather than partitioning it into subgraphs as with such storage needless transfer of data can be avoided that may take place in case of partitioning and distributing a graph over multiple machines. The authors combine adhoc indexes over the evaluation algorithms that rely on these indexes for computing expensive join operations.

Table 4, compares the frameworks storing data in partitioned files on Hadoop or in local storage or using a single table scheme for Big RDF data storage.

3.2. Binary Table (or Vertical Partitioning)

Husain et al. [54] proposed a scheme for storing Big RDF data on the HDFS and an algorithm for generating best possible query plans for answering SPARQL queries based on a cost model. The whole RDF data is not stored in a single file and instead this data is divided into multiple smaller files. The data is first split, on the basis of predicate (predicate split), next it is split based on the predicate object (predicate object split). Here, the MapReduce framework is used for answering the queries where the map phase is used for selection while the reduce phase is used for join.

Tanimura et al. [103] proposed a RDF system that incorporates Pig into its architecture. The three interfaces of this system are: general data processing interface, interface for RDF data processing and a custom data processing interface. The storage schema is defined at the lower layer and Pig's query engine is used for implementing the optimization using this schema. This system uses the vertical partitioning (VP) scheme for storing Big RDF data into tables.

Husain et al. [53] proposed a storage scheme for storing Big RDF data in the HDFS. The authors presented an algorithm for generating the query plan which uses a greedy approach for answering SPARQL queries. In this framework, the two partitioning components applied are; Predicate Split (PS) and Predicate Object Split (POS). The RDF data is split into predicate files using this PS component. After this step, these predicate files are given as input to the POS component. The predicate files are divided into smaller files based on the type of objects by this POS component. During data preprocessing the RDF data is processed and populated in files in the HDFS. The data files are partitioned and organized also, dictionary encoding is executed during the data preprocessing process. A scalable framework for semantic web constructed using cloud computing technologies is proposed by authors in [55]. The authors devised a novel algorithm for handling complex SPARQL queries. The proposed framework consists of 2 major components i.e. data preprocessing and query answering.

Schätzle et al. proposed PigSPARQL [97] which translates the SPARQL queries into Pig Latin. The Pig

Author	Compared/ Tested with	Evaluation metrics	Tested dataset(s)	Weakness/Limitation(s)
Huang et al. (2011) [52]	SHARD [93], RDF-3x [78], Hash partitioning	Data load time, Query execution time	LUBM	- Where MapReduce is used for partial result aggregation a significant degrade in performance is observed when queries exceed the n-hop guarantee.
Lee et al. (2013) [62]	Random partitioning (rand), simple hash partitioning on subject (hash-s), graph partitioning, hash partitioning on both subjects and objects	Data partitioning & loading time, Data redundancy Data distribution, Query execution time, Scalability, Effects of optimizations	LUBM, SP2Bench, DBLP, Freebase	- The cost of join evaluation may be higher as original and replicated data is managed in same set of indices thus, resulting in duplicate and large intermediate results.
Galárraga et al. (2014) [34]	RDF-3x [78], Predicate-based partitioning, Huang et al. [52]	Queries per second (Throughput)	BTC 2008	- Here, the regular query workload needs to be known prior to query execution and the variation in query workload over time can result in suboptimal partitions.
Wang et al. (2015) [105]	Husain et al. [53], Huang et al [52]	Partitioning & loading time, Query execution time, Data redundancy, Intermediate results	LUBM, BTC	- Before the data is distributed to respective nodes significant data pre-processing and additional data structures are required.
Leng et al. (2017) [64]	Two graph partitioning algorithms i.e. METIS, MLP+METIS and a hash partitioning algorithm	Query execution time, Scalability	LUBM, SP2Bench, DBLP	- It takes more time than hash partitioning for star pattern queries in 2-hop guarantee
Goasdoué et al. (2013) [35]	Husain et al. [53]	Data upload time, Query execution time, Network traffic (shuffled data)	LUBM	- The MapReduce joins can be expensive thus resulting in a cost overhead.
Goasdoué et al. (2015) [36]	SHAPE [62], H2RDF+ [85]	Query execution time	LUBM	- It replicates the whole dataset three times.
Chebotko et al. (2012) [18]	Different SPARQL queries	Query results, Query execution time, Scalability	UTPB	- The simple queries even though involving no joins take more time because of the time taken to transfer final results to client machine.

Table 4

Summary of Big RDF frameworks using Partitioned files in local storage/hadoop or single table storage scheme

Latin programs thus obtained are executed as a series of MapReduce jobs on the Hadoop cluster. PigSPARQL uses the Vertical Partitioning (VP) strategy for reducing the number of RDF triples that need to be loaded at the time of query execution. A single MapReduce job is sufficient for doing VP and the VP can be done in advance as it doesn't cost extra disk space. A standard approach is used in this framework to translate SPARQL queries into Pig Latin. An abstract syntax tree (AST) is generated after parsing the SPARQL query, next this AST is translated into a SPARQL algebra tree.

Du et al. [29] proposed HadoopRDF that combines Hadoop and the traditional triple stores. Thus, a triple-store i.e. Sesame is installed here on each node in the cluster. A part of RDF data is stored in Sesame so that it becomes possible to execute flexible SPARQL queries on nodes in the cluster. The RDF dataset is partitioned and stored in Sesame on separate nodes. In HadoopRDF, Hive tables are created to store the triples and SPARQL queries are written into HQL's. The HQL queries are thus executed on the Hive tables.

A framework proposed by Ali et al. [8] for storing Big RDF data leverages the cloud computing paradigm.

This architecture consists of two components i.e. data pre-processing and query answering. The RDF data stored in a N-triples file is split into predicate files by the PS module. The POS module then takes as input these predicate files and splits them into smaller files on the basis of the type of objects. The Big RDF data is maintained in flat files on HDFS in this framework. The intermediate results obtained during query processing are saved into logical disk. This is done to accelerate the processing of similar queries that may later come into view.

Zhang et al. [114] proposed a RDF join processing solution that is based on a cost model for minimizing the response time of queries to maximum possible extent. Here, the Big RDF data is first decomposed into the predicate files and is then organized according to the data contents. The Big RDF data is stored on HDFS and the query engine in proposed system accepts SPARQL queries from the users, decides the corresponding MapReduce jobs and an optimal query execution plan for each query.

Wu et al. [107] proposed a technique to place RDF triples according to the relationship between them using a schema based partitioning strategy. This framework is based on the purpose of reducing number of MapReduce cycles required during a job for each SPARQL query and of storing related triple on the same node. This technique is prototyped in HadoopDB along with the columnar database MonetDB. In the proposed approach, the database i.e. in HadoopDB jobs, SQL expressions are obtained after conversion of SPARQL sub-queries while in HDFS jobs, reduce-side join method is used for executing joins in queries.

Li et al. [65] designed a vertical partitioning like model to store Big RDF data in HBase. This method requires less space and is more efficient for query processing. According to the proposed storage schema, two predicate tables i.e. PSO and POS are used for storing the RDF triples. There is only one column family for each table. The subject of a RDF triple is selected as row key, table name uses the predicate of this triple and object is represented as a value in the cell in this PSO table. In similar manner, object of the RDF triple represents the row key, subject of this triple is represented by the cell value and table name uses the predicate of the triple in a POS table. A path index is also proposed in this storage model for reducing the cost of multi-table joins in case of complex queries.

Graux et al. proposed SPARQLGX [38] that translates SPARQL queries into executable code which is evaluated over Big RDF data in accordance to the method of storage that is being used and computed data statistics. This frameworks leverages advantages of the Vertical Partitioning strategy such as no complex computation, data size reduction and indexing (as it limits the search only to a few relevant files).

Schätzle et al. proposed S2RDF [99] for RDF Querying with SPARQL on Spark that overcomes the drawbacks of other Big RDF frameworks which favor certain query shapes. It does so by minimizing the input size of query without taking into regard its shape and diameter. The data layout proposed in S2RDF is complementary to the vertical partitioning (VP) approach as this layout also distributes triples into relations comprising of two columns (where one is for subject and second one is for object) corresponding to the RDF properties.In this framework, an extension of Vertical Partitioning schema is proposed i.e. ExtVP (Extended Vertical Partitioning) where dangling tuples in a query are avoided to a large extent. S2RDF uses an optional method for storage optimization where a selectivity threshold can be defined for ExtVP. This helps in minimizing the size overhead as compared to VP while maintaining its performance benefits.

Madkour et al. [71] proposed SPARTI (Scalable RDF Data Management Using Query-Centric Semantic Partitioning) which employs a novel relational partitioning schema known as SemVP. Similar, to the vertical partitioning (VP) scheme SPARTI partitions the RDF data based on property and stores the subject and object of a property in a SemVP. This SemVP extends each entry with the row-level semantics named as the semantic filters. These semantic filters indicate that whether an entry in a partition is the part of a query join result or not. Here, the quality of partitioning scheme is assessed by a cost model and a budgeting mechanism. The Parquet columnar store format is used for storing RDF data in this framework.

Hassan et al. [48] proposed two approaches i.e. 3CStore and VPExp on the basis of the existing vertical partitioning approach. In VPExp, the predicates are split using the explicit type of the object while, in 3CStore sub-tables are created from VP table on the basis of different join co-relations that exist between triple patterns. VPExp minimizes the data size of input for the predicate rdf:type during query evaluation. This strategy is useful for queries with triple patterns

having a rdf:type predicate and a non-variable object. 3CStore is a three column layout that precomputes a subset of VP table based on the subject-object (SO), subject-subject (SS) and object-subject (OS). The join operations and cost of communication for SPARQL queries in distributed systems can be minimized using 3CStore approach.

3.3. Property Table

Schätzle et al. proposed Sempala (a SPARQL-over-SQL-on-Hadoop) approach [98] for selective SPARQL queries. In Sempala, the Big RDF data is stored on HDFS in a columnar layout and Impala forms the execution layer for processing SPARQL queries translated into SQL expressions on Hadoop. Thus, a RDF data layout that uses Parquet (a columnar storage layout for Hadoop) and thus, is space efficient for Impala is proposed by the authors. Sempala uses a single unified property table that contains all the properties contained in the RDF dataset for reducing number of joins that are required by a query. The RDF data in N-Triples format is converted into this unified property table layout by using MapReduce. It is observed that this layout achieves an excellent compression ratio in comparison to Triple table and Vertical Partitioning (VP).

3.4. Mixed

Cossu et al. proposed PRoST (Partitioned RDF on Spark Tables) [24] that stores Big RDF data using a hybrid scheme by combining two popular RDF storage techniques i.e. Binary and Property tables (Vertical Partitioning+Property Table). In ProST, the data is partitioned in two different ways and thus it is stored twice. This is done because different query types are benefited by a different storage approach. PRoST overcomes the drawback of large number of NULL values in Property table by storing the Big RDF data in Parquet. Also, the other problem with Property tables of multi-valued properties is overcome in ProST by storing the object data in lists which can be flattened at the time of execution.

Hassan et al. [49] proposed techniques for distributed storage on No-SQL based systems i.e. HBase and Cassandra and in-memory processing i.e. Spark SQL for Big RDF data management. The property table and vertical partitioning schemes are used for storage on HBase and Cassandra respectively. A data loader is presented for loading data in HBase and Cassandra.

Table 5, summarizes the frameworks that use a binary table or property table or a mixed (a combination of binary and property table) scheme for Big RDF storage.

3.5. Graph-based

Mutharaju et al. proposed D-SPARQ [75], a distributed query engine which combines a NoSQL store i.e. MongoDB with the MapReduce processing framework. Here, a single MapReduce job is used to import the received Big RDF datasets into MongoDB. This MapReduce job also captures the necessary statistics that are required by the join reordering module during the query optimization process. D-SPARQ stores all triples with same subject in one file.

Zeng et al. proposed Trinity.RDF [112], a distributed in-memory system for managing Big RDF data. The Big RDF data is stored in its native graph form in memory and the RDF data is modeled as an inmemory graph. This graph exploration replaces the costly join operations and is performed on all distributed machines in parallel. Trinity.RDF is based on Trinity, a distributed inmemory key-value store and on top of this key-value store it builds a graph interface. A custom communication protocol that is based on the Message Passing Interface (MPI) standard is used in this system.

Hose et al. [50] proposed WARP (Workload-Aware Replication and Partitioning for RDF) a distributed SPARQL query engine which combines the popular partitioning technique i.e. graph partitioning with the workload aware replication of triples. The aim of this system is to push as much as possible execution of queries into the centralized RDF store. The query workload is taken into consideration here for choosing the RDF data parts to replicate. As a result, there is no requirement of replicating rarely used RDF data thereby, resulting in low storage overhead.

Chen et al. [19] proposed SparkRDF, an elastic discreted RDF graph processing framework with distributed memory. It splits the RDF graph into small multi-layer elastic subgraphs (MESG) based on the classes (C) and relations (R) which helps in cutting down the search space and avoids memory overhead. This framework also supports fast iterative join operations by caching the Resilient Discreted SubGraph (RDSG) modeled intermediate results into distributed memory. The architecture of SparkRDF consists of five modules i.e. preprocessing of RDF data, splitting the RDF graph, distributed storage, parsing queries and distributed join.

Schätzle et al. [95] proposed S2X, a SPARQL query processing framework for Big RDF data. In S2X the Basic Graph Pattern (BGP) matching for SPARQL queries is implemented in a graph parallel manner unlike the other operators which are computed in a data manner. The graph pattern matching part in SPARQL is implemented using the graph parallel abstraction of GraphX while the data parallel computing is implemented using Spark. In S2X the RDF data is mapped to the property graph model of GraphX.

Balaji et al. [14] proposed a novel graph path query processing distributed system based on the Apache Spark framework. The authors introduce a graph data model for distributed processing and provide mechanisms for querying graph path queries using the Spark framework. In contrast to the other systems, this system doesn't require to build extensive indices. All the matching graph substructures that can satisfy an issued query are identified during the process of graph path querying. In the proposed framework, no heavy graph level indexes need to be built.

Gombos et al. proposed Spar(k)ql [37] which uses GraphX to evaluate SPARQL queries on the Big RDF dataset in a distributed fashion. This distributed graph processing system uses message passing to pass messages between the nodes for calculating final result. Here, the vertex program sends messages to the appropriate neighbors in each iteration. The query generator in Spar(k)ql determines the messages which are necessary to answer the query. During the first step, the RDF graph is loaded into memory. Also, the nodes with their properties are stored in one file while the edges are stored in an another file.

Bahrami et al. [13] proposed a novel approach of query optimization for processing SPARQL queries efficiently over GraphFrames API. Here, the GraphFrames API based on the Apache Spark framework is used to cut down the query search space and SPARQL queries are optimized by reordering triple patterns according to some ranking criteria. The workflow of the proposed approach comprises of 5 modules. The first module i.e. Data Graph Generation scans the entire RDF dataset and 2 separate lists i.e. edgeList and nodeList are generated and stored in CSV files.

Kassaie et al. [59] uses the graph view of GraphX and collection view of Spark for iterative pattern matching and merging partial results of the subgraph matching respectively. The authors proposed a parallel subgraph matching algorithm where the RDF data is represented as a graph using RDD in GraphX. Here, each graph vertex is assigned three properties i.e. a label to hold the value of its corresponding subject or object, a Match_Track table (M_T) to collect variable and constant mappings and an end flag to indicate a vertex that is located at the end of a path.

3.6. Hybrid

Gurajada et al. [41] proposed Triple-Asynchronous-Distributed (TriAD) which is based on a novel shared nothing main memory architecture and the asynchronous message passing protocol. This system employs join executions that are asynchronous using a custom Message Passing Interface (MPI) protocol along with a lightweight join-ahead pruning (i.e. pruning of triples that might not qualify for a join) technique for distributed SPARQL query execution. This system follows a traditional master-slave architecture where the slave nodes operate in an autonomous manner. The slaves communicate directly through the messages that are exchanged asynchronously for running multiple join operators along the query plan in parallel. Abdelaziz et al. [3] suggested that specialized in-memory systems like AdPart [44] and TriAD [41] give the finest performance.

Wylot et al. proposed DiploCloud [108], a scalable and efficient RDF framework in the distributed and cloud environments. This system uses a hybrid storage model to efficiently partition the RDF graph and co-locate related instance data. A novel data placement technique is used in this system for co-locating the pieces of data that are semantically related. The proposed strategies for loading data and executing SPARQL queries take advantage of the partitions and indices of the system. These co-located data patterns are extracted from both the schema and instance levels. The three main structures in this system include template lists, molecule clusters and molecule index. The DiploCloud architecture, contains a master node which comprises of three main subcomponents i.e. a partition manager, a key index for encoding URIâĂŹs into IDâĂŹs and a distributed query executor.

3.7. Any

Hammoud et al. [42] proposed DREAM, that combines the benefits of both distributed and centralized systems. It does so by avoiding RDF data partitioning

Author	Compared/ Tested with	Evaluation metrics	Tested dataset(s)	Weakness/Limitation(s)
Husain et al. (2010) [54]	Jena In-Memory, Jena SDB, BigOWLIM	Query execution time, Scalability	LUBM, SP2Bench	- The usage of a reduce-side join approach results in sorting and shuffling overhead.
Husain et al. (2011) [53][55]	Jena In-memory, Jena SDB, BigOWLIM, RDF-3x [78]	Query execution time, Scalability	LUBM, SP2Bench	- The join selectivity is not taken into consideration by the greedy planner. - A large overhead is induced for selective queries as joins in that case are executed only with MapReduce jobs.
Schätzle et al. (2011) [97]	optimized, optimized+partitioned, non-optimized+partitioned queries	Query execution time, Scalability, HDFS bytes read, written & reduce shuffle bytes	SP2Bench	- The batch oriented nature of MapReduce causes relatively high query latencies.
Du et al. (2012) [29]	Different SPARQL queries, Hive	Data load time, Query execution time, Scalability	BSBM	- It is not efficient for complex queries as a lot of time is consumed to output final results so I/O delay consumes most time.
Ali et al. (2012) [8]	Sesame+DAMLDB, Jena+DAMLDB	Query execution time	LUBM	- Unlike DAMLB, it doesn't use any indexing techniques for improving query performance.
Zhang et al. (2012) [114]	Three join strategies	Query execution time	BTC	- The storage scheme used works well only in case of triple patterns having a bound predicate.
Wu et al. (2012) [107]	Sideways information passing (SIP), Schema based hybrid partitioning (Hybrid), SIP+ Hybrid, RawHadoop, Hive, Hash partitioning	Data distribution over cluster (ratio of data size in a node to the total data size), Intermediate results size, Query execution time, Scalability	LUBM	- It pushes as much join processing as possible on a single node by gathering related triples at one computing node thus, resulting in load imbalance.
Li et al. (2015) [65]	Rya [91], Jena 2	Storage space, Data ingest time, Query execution time, Scalability	LUBM	- The RDF triples are repeatedly stored in this storage schema thus resulting in a wastage of lots of storage space. - The path index find it hard to query the results of intermediate variable at the same time. - It is very inconvenient to manage these tables if, there are tens of thousands of predicates.
Graux et al. (2016) [38]	Rya [91], CliqueSquare [35], S2RDF [99], RDFHive, PigSPARQL [97]	Preprocessing time, Disk footprint, Query execution time	LUBM, WatDiv	- It needs to read entire data in order to compute statistics over dataset. Instead, it should take the query structure into account and compute statistics only on query related data.
Schätzle et al. (2016) [99]	SHARD [93], PigSPARQL [97], Sempala [98], H2RDF+ [85], Virtuoso	Data load time, HDFS size, Query execution time	WatDiv	- It suffers from the high cost of data loading and preprocessing. It trades off the performances with disk space and loading time. The startup costs limits its applicability to Big RDF data. - The semi-joins are expensive to compute and generate large network-traffic.
Madkour et al. (2018) [71]	Vertical partitioning (VP), S2RDF [99]	Storage space, Query execution time, Data load time, query workload	WatDiv, Yago	- It also computes reductions like S2RDF [99] for improving query performance. This preprcoessing can impose a lot of overhead.
Schätzle et al. (2014) [98]	Hive, PigSPARQL [97], MAPSIN [96], MapMerge	Data load time, Data storage size, Query execution time	LUBM, BSBM	- It is not adapted to other query shapes and is only efficient for star-shaped queries.
Cossu et al. (2018) [24]	Vertical partitioning, SPARQLGX [38], S2RDF [99], Rya [91]	Data load time, Data storage size, Query execution time	WatDiv	- The data is stored twice using two different storage schemas i.e. PT and VP. It occupies double space than SPARQLGX [38].
Hassan et al. (2018) [49]	compares Hbase & Cassandra	Data load time, Query execution time	BSBM, SP2Bench	- The bulk loading of Big RDF data in Hbase and Cassandra can be cumbersome task
Hassan et al. (2018) [48]	Vertical partitioning (VP), S2RDF [99]	Data load time, Storage space, Query execution time	LUBM WatDiv	- The proposed 3CStore data layout outperforms the proposed VPExp layout, standard VP layout and S2RDF system. But it requires more storage space with a longer data loading time

Table 5

Summary of Big RDF frameworks using binary table or property table or mixed storage scheme

and thus partitions SPARQL queries only thereby avoiding shuffling of intermediate data. In this system, the complete RDF dataset is stored on each machine in the cluster and a query planner is employed to effectively partition any SPARQL query. A rule and cost-based planner is used in DREAM which uses the statistical information of RDF data. DREAM adopts a master-slave architecture and RDF-3x deployed on each slave machine.

In Table 6, we present a summary of the frameworks using a Graph-based or Hybrid or Any sche-me for Big RDF storage.

Discussion

We observed that most Big RDF frameworks store data in index tables or use the Binary table format for storage. The benefit with these formats is that they support efficient data retrieval. After storing the data in index tables some of the SPARQL queries can be easily resolved by scan of a single table. The binary table format is also efficient for queries where the predicate is not a variable (which is true in most SPARQL queries). Thus, only a single binary table needs to be accessed for each triple pattern in the SPARQL query. These two data storage formats improve query performance by reducing number of table scans.

4. Big RDF Preparation

RDF data partitioning and indexing are two prime components of a Big RDF framework that need to be upgraded for facilitating efficient retrieval on Big RDF data. We discuss the various techniques used by these components in a Big RDF framework in this section.

4.1. Big RDF Partitioning

Curé et al. [27] compared five RDF distribution approaches by implementing them over Spark API. The approaches which are compared are two hash based i.e. Random hashing and RDF triple element hashing, 2-graph partitioning based i.e. Huang et al. (denoted as nHopDB) [52] and WARP [50] and a hybrid approach. The final hybrid approach combines the hash-based partitioning and query workload aware processing. After experimental evaluation it can be concluded that the hash-based partitioning approaches are more effective than the graph-based ones. Also, the hybrid and hash-based approaches are two finest schemes and are close to each other.

4.1.1. Hash Partitioning

SHARD [93] hash partitions the RDF triples by subject across multiple machines and parallelizes the query processing. The use of this kind of partitioning allows SHARD to optimize the subject-subject joins. SARROD [104] persists the Big RDF data as basic RDF triples and SPARQL queries are run over this data. The authors used SHARD triplestore for query execution and evaluation of algorithm in SARROD. So, SARROD also takes leverage of the partitioning technique used by SHARD i.e. hash partitioning by subject.

SHAPE [62] uses a semantic hash partitioning method during the data partitioning phase. The access locality is utilized in this framework to partition Big RDF graphs across multiple nodes. The intra-partition processing capability is maximized and inter-partition cost is minimized for the same. This framework uses hop-based triple replication to create a set of semantic hash partitions such that the number of queries that can be evaluated by intra-partition processing is increased and maximized.

Rainbow [39] achieves dynamic scalability by partitioning Big RDF data in the memory storage using a consistent hashing algorithm. Thus, the data stored in the Redis stores is partitioned using this algorithm with good load balancing and scalability. Here, the space of hash keys are arranged in a ring and hashing is used for mapping both the hosts as well as data on the ring.

JARS [92] uses a dual hash partitioning of triples on the basis of subject and object alongwith a two-layered distributed clustered indexing and a rule-based query execution approach. The database in each server consists of two tables, the first for storing triples with same hash value for subject and second, for holding triples with same hash value for object. Thus, in this framework the triples are stored twice.

The method proposed [77][76] in uses a hash-based query independent partitioning strategy to partition and distribute Big RDF data in the cluster. All the triples in the input dataset are partitioned on the hash value of their subject. The decision of processing a join locally or evaluating it by transferring data between nodes is made according to the partitioning scheme.

The strategy used for partitioning in [29] is that the triples having same predicate are allocated into the same part of data (Hash-based partitioning on predicate) and are distributed in a balanced manner. The RDF data is randomly partitioned across cluster by hashing on nodes in Trinity.RDF [112]. The framework pro-

Author	Compared/ Tested with	Evaluation metrics	Tested dataset(s)	Weakness/Limitation(s)
Mutharaju et al. [75]	RDF-3x [78]	Query execution time	SP2Bench	- The initial RDF partitioning can become a bottleneck with increasing data size as centra-lized METIS graph partitioner is used to partition input RDF dataset.
Zeng et al. (2013) [112]	RDF-3x [78], Bitmat [12], Huang et al. [52]	Query execution time, Machine & data scalability, Query result size, Storage space	BTC, Dbpedia, LUBM	- Its performance is bound by the main memory capacity of the cluster, as the whole set of triples needs to be loaded in main memory.
Hose et al. (2013) [50]	RDF-3x [78], Huang et al. [52] (with no MapReduce)	Query execution time	BTC	- It only works well for static query work-loads. It adapts only by applying expensive re-partitioning of the entire data; otherwise, it incurs high communication costs for dynamic workloads
Schätzle et al. (2015) [95]	PigSPARQL [97]	Query execution time	WatDiv	- The hash-based encoding has very long loading time and cannot load all graphs, such as YAGO2. On the other hand, count-based encoding has faster data loading in GraphX but is slightly slower in query runtime.
Gombos et al. (2016) [37]	S2X [95]	Query execution time	LUBM	- It suffers with an overhead due to existence of loop edges, which requires a number of messages to be passed. It doesnâĂŹt do the analysis of ordering of triple patterns.
Kassaie et al. (2017) [59]	Different SPARQL queries	Query execution time, Scalability	LUBM	- The drawback of this technique is lack of query optimization via optimal evaluation of BGP triples
Gurajada et al. (2014) [41]	RDF-3x [78], Bitmat [12], SHARD [93], 4-store [46], Trinity.RDF [112], Huang et al. [52]	Scalability, Communication cost, Query execution time	LUBM, WatDiv, BTC	- It induces significant cost for queries producing large intermediate results with multiple attributes.
Wylot et al. (2016) [108]	Virtuoso, RDF-3x [78], 4-store [46], Jena [73], BigOWLIM, AllegroGraph, SHARD [93], Huang et al. [52]	Data load time, Storage space, Query execution time, Scalability	Bowlog-naBench, LUBM, Dbpedia	- This approach makes more interprocess traffic, given that related triples, winds up being scattered on all machines.
Hammoud et al. (2015) [42]	RDF-3x [78], H2RDF+ [85], Huang et al. [87]	Network traffic, Query execution time, Scalability	LUBM, Yago	- The resources of a single node can become a bottleneck as the whole dataset must be loaded into RDF-3X on every node. - The excessive replication limits its applica-bility to large scale RDF data.

Table 6

Summary of Big RDF frameworks using Graph-based or Hybrid (Triple+Graph-based) or Any storage scheme

posed in [14] uses a hash partitioning strategy for vertex partitioning. SparkRDF [19] uses a hash partitioning strategy to hash partition RDF triples on the subject.

4.1.2. Graph Partitioning

In [52] the triples are partitioned using a graph partitioning algorithm so the triples that are close together in RDF graph are stored on the same machine. Thus, a smaller amount of network communication is incurred at the time of query execution. The data partitioner at master uses the graph partitioning algorithm to execute the disjoint partitioning of input RDF graph by vertex. The METIS partitioner is used here for partitioning the input RDF graph.

A graph partitioning mechanism is used for distribution of Big RDF data across the workers in EAGRE [113]. Also, an in-memory index structure is implemented for efficiently accelerating evaluation of the order sensitive and range queries. The structure locality of original RDF graph is preserved by partitioning the entity classes in a suitable manner. EAGRE uses METIS for partitioning the global compressed entity graph. Next, these entities are placed in accordance to the partition set to which they belong.

Oh et al. [81] uses the METIS partitioner for RDF graph partitioning. The RDF graph is compressed into its abstract version for creating an efficient query plan and the partitioning information is used for compressing it. The partition information is encoded into the schema so adjacent RDF data can be loaded into the same HBase region. This information is used to create a filter that further reduces the input size to the map jobs. The authors use a approach similar to proposed by Huang et al. [52] where partitioning information is used for assigning the adjacent RDF data to same machine in a cluster. A similar approach is used by TriAD [41] for pruning dangling tuple prior to query processing.

4.1.3. Query Workload-aware Partitioning

Partout [34] relies on a partitioning based on query workload such that the SPARQL queries can be executed over a minimal number of nodes. This system analyzes the query workload for collecting information about the frequently co-occurring subqueries. Here, the process of partitioning is divided into two tasks i.e. fragmentation (triples are split into pieces) and fragment allocation (storing each piece into a node). In fragmentation, the triple relations are partitioned horizontally on the basis of the set of constants appearing in queries. To ensure the local execution of most queries at a host the fragment allocation is done while, simultaneously maintaining load balancing.

The framework proposed in [105] partitions Big RDF data according to query workload and the partitions are physically placed to reduce data redundancy. This system partitions RDF data according to their query patterns thus, ensuring that no cross node join operations occur when queries are evaluated according to any registered pattern. Each query partition would have a separate partitioning and a procedure of data placement will be used for putting the partitions generated by all recurrent query patterns into nodes. When the original RDF data is partitioned in accordance to a query pattern Q, the triples not satisfying this pattern are not allocated to any partition. There are many such kind of triples and thus, these are not partitioned. Such type of triples are accessed rarely and thus, they are known as cold triples.

In [10], the tables are partitioned in the cluster based on join attributes. Then, the SQL query is executed on these partitioned and distributed RDDâĂŹs and finally the resultant table obtained is the final output. The authors compare the approach for two types of partitioning first, where the data is partitioned on an irrelevant field and second on the fields that are used in SQL query. The proposed approach for these two types of partitioning is tested on a single node and in a cluster.

SPARTI [71] partitions RDF data by analyzing the query workload. Thus, Big RDF data is partitioned by analyzing the join patterns that are found in the query workload. Here, the performance of frequent join patterns is improved by vertically partitioning (VP) Big RDF data and then updating the partitioning in an incremental manner by analyzing the query workload. The partitioning scheme used here is termed as SemVP (Semantic Vertical Partitioning), in this scheme a reduced set of rows are read instead of the complete partitions.

4.1.4. Horizontal Partitioning

The problem of data organization is handled by SHOE [66] by importing a mapping for identification along with the horizontal partitioning scheme. The data files of the partitioned triples are maintained on HDFS. The objective of this type of partitioning is to use simple range filtering for reducing the actual cardinality of each MapReduce job.

In [107] firstly, the RDF data is analyzed for extracting the schema in various scenarios to specify relationships. After the schema is extracted and the relationships are obtained the complete RDF data is decomposed into numerous partitions and the related triples

are maintained in the same partition. The proposed partitioning method combines the binary table (vertical partitioning) storage method and the horizontal partitioning strategy while simultaneously considering schema relationships.

For achieving maximum benefit from property table approach ProST [24] horizontally partitions Big RDF data based on the subject.

4.1.5. Hybrid

The framework proposed in [28] considers load balancing during data placement process. During the query evaluation the queries are rewritten into small tasks based on partitioning and placement strategy and are run in parallel over different logical partitions. A simple re-partitioning strategy of hashing the RDF triples on their subjects is used to divide large partitions into small segments. The workload is balanced across nodes by using a round robin assignment technique.

Initially, a lightweight hash partitioning is applied in AdHash [43] for distributing RDF triples by hashing on their subjects. This type of partitioning is favorable for the parallel processing of triple patterns with join on their subjects without any data communication. This system adapts dynamically to changing workloads to overcome the limitations of other Big RDF systems which use a static partitioning scheme such as WARP [50] and Partout [34]. In this framework, the query workload is monitored and a hierarchical heat map of the accessed data patterns is incrementally updated.

During the first step in [26], fragmentation is done by hash partitioning Big RDF data using subject as key. Thus, this supports local execution of star-shaped SPARQL queries but it provides no assurance for other shape queries such as chain, tree, cycle etc. The data allocation is done by analyzing frequently executed queries over the dataset. The tasks of hash-based partitioning and query workload aware distribution are implemented as Spark programs to ensure their parallel execution.

The framework proposed in [4] uses the classical predicate, subject based partitioning techniques in combination with the knowledge of query workload. This combination helps in maximizing intra node execution of chosen queries. The incoming data is partitioned incrementally using the machine learning techniques. The Random Forest and Logistic Regression algorithms are used for classifying RDF data and the efficiency of these algorithms is assessed by using the procedure of incremental partitioning.

In D-SPARQ [75], a graph is constructed from the given RDF triples and the triples are divided across nodes in the cluster using a graph partitioner (number of data partitions are equivalent to number of machines). The triples are places into partitions by matching them to vertex. Thus, if the subject of a triple matched against the vertex then it is placed into same partition as the vertex. Thus, it can be seen that hash partitoning by subject is applied in DSPARQ similar to the one applied by Huang et al. Here, MongoDB stores the triples that are assigned to each partition or machine. This framework imports the received RDF datasets into MongoDB using a single MapReduce job.

WARP [50] uses METIS graph partitioner for partitioning the RDF graph. Each RDF partition is stored in a dedicated centralized store i.e. RDF-3x. By using such partitioning each triple pattern in the query is evaluated by shipping it to all the hosts where it is evaluated on the local data in a parallel manner and the results are returned to the query initiator. WARP which is influenced by Huang et al. [52] (denoted as nHopDB) as well as Partout [34], borrow the graph partitioning and 2-hop guarantee from nHopDB and then similar to Partout the triple allocation is refined by taking the query workload into consideration.

TriAD [41] uses a locality-based and horizontal partitioning scheme for partitioning the triples. The SPO permutation list six indexes in this framework are first hash partitioned on its join key and are then stored lexicographic order. The partitioner in this framework partitions the incoming RDF triples to create the summary graph using a non-overlapping graph partitioning algorithm such as METIS. So, the partitioner is responsible for partitioning triples using graph partitioning and local SPO index structures using hash partitioning in this framework.

4.1.6. Other

The built-in data replication mechanism of Hadoop Distributed File System (HDFS) is exploited by CliqueSquare [35]. By default each partition has three replicas as RDF dataset is partitioned in different ways. For first replica, the triples are partitioned based on their subject, property, and object values. In CliqueSquare, all subject, property, and object partitions of the same value are stored within the same node for the second replica. Finally, all the subject partitions within a node are grouped by the value of property in their triples for the third replica. Similarly, in CliqueSquare all object partitions are grouped based on their property values.

In CSQ [36], partitioning and placement of triples is done based on their subject, predicate and object values such that they are located on same compute node. These are called as subject, predicate and object partition and for subject-subject (SS) joins this local partitioning limits its access to only few triples. Further, each of this partition inside a compute node is partitioned on the basis of its property value. Finally, this each resulting partition is stored in a HDFS file by using the value of property as filename. In RQCCP [40], the RDF data is split and stored in HDFS by its class of Subject, Predicate and the class of Object.

A novel relational partitioning schema called ExtVP is proposed in S2RDF [99]. In ExtVP, the fundamental idea is to pre-compute a number of semi-join reductions for a vertically partitioned table. This schema that uses a preprocessing based on semi-joins for reducing the input size of the query significantly. The relevant semi-joins between the vertically partitioned tables can be determined by finding the feasible joins i.e. Subject-Object (SO), Subject-Subject (SS), Object-Object (OO) and Object-Subject (OS) join. The joins that may occur when joining results of the triple patterns at the time of query execution. It uses the ExtVP tables if they exist otherwise it the normal VP tables are used.

Discussion

We observed that hash partitioning is one of the dominating approaches in RDF data partitioning as it is more suitable for RDF data structure. It can be seen that the Big RDF frameworks using hash partitioning commonly hash the triples by their subjects as it guarantees that the star queries can be locally evaluated. In comparison, to the other partitioning schemes, hashing incurs no storage overhead.

4.2. Big RDF Indexing

The framework proposed in [60] compares different storage layouts with different indexing schemes. The authors compared layouts such as Simple, Indexed, Vertically Partitioned, VP+Indexed, Hybrid (Simple+VP) and Hash. The indexed layout uses the six-table index scheme where six tables i.e. SPO, PSO, OSP, SOP, POS and OPS that represent the six possible combinations of triples are constructed. In the VP+Indexed layout, SPO, OS and OSP additional tables are constructed and stored in HBase.

4.2.1. Eight-index scheme

The six indexes materialized by H2RDF+ [85][87] are SPO, PSO, OSP, SOP, POS and OPS that are maintained in HBase tables. Apart from this, the two categories of aggregated index statistics materialized for estimating triple selectivity, join output size and join cost are: SP_O, PS_O, PO_S, OP_S, OS_P and SO_P and, S_PO, P_SO, P_OS, O_PS, O_SP and S_OP. So, in total H2RDF+ maintains eight indexes (six index+two aggregated indexes).

4.2.2. Six-index scheme

The storage structure proposed in [102] takes into consideration characteristics of both the RDF model and the storage structure of HBase. The six tables built to store the RDF triples are S_PO, P_SO, O_SP, PS_O, SO_P and PO_S and in these tables row keys and column names. The advantages of this data storage structure is that it supports multivalued properties, supports parallel processing and reduces the I/O cost. The disadvantages are that requires more storage space and complicated update operations.

The authors in [18] define several bitmap indices to speed up the query evaluation. The indices are denoted as I_{ss}, I_{oo}, I_{so} called as the join indices while other indices are I_s, I_p, I_o that are called as selection indices. The selection indices can help in speeding the matching process of distinct triple patterns while the join indices are used for joining the intermediate results obtained through subgraph triple pattern matching.

ScalaRDF [51] stores six indices i.e. SP_O, SO_P, PO_S, P_SO, O_SP, S_PO key-value pairs and the consistent hashing protocol is used to distribute these indices among the slave nodes. Here, the index is stored in memory for improving performance of queries. TriAD [41] employs six primary SPO permutation indexes i.e. object key indexes (OSP, OPS, POS) and subject key indexes (SPO, SOP, PSO).

4.2.3. Five-index scheme

SparkRDF [19] creates five kinds of indexes i.e. C (Class subgraphs), R (Relation subgraphs), CR, RC and CRC (combinations of class and relation subgraphs).

4.2.4. Four-index scheme

In CumulusRDF [61], three indexes i.e SPO, OSP and POS are used to satisfy all RDF triple patterns and provide a fast lookup for all triple patterns. These indices support a complete index on the triples and the lookups on named graphs (contexts). The standard key-value model of Cassandra is used to store these indices

in a flat layout. CumulusRDF uses a secondary index i.e. CSPO to map the column values to row keys, The property-object entries associated with a given property value can be retrieved using this index.

Trinity.RDF [112] employs a four index scheme where apart from the two indices i.e SPO and OPS the two other indices built are PS and PO. The graph exploration used in this framework retrieves nodes of a given predicate that are connected by an edge. This PS and PO index during graph exploration enables to find all the outgoing or incoming neighbors labeled by a given predicate. Thus, a machine stores a key-value pair for each predicate.

4.2.5. Three-index scheme

In AMADA [9], the data indexes are implemented within SimpleDB which supports SQL-style queries based on the key-value model. From the system architecture of AMADA it can be seen that it stores dataset in form of a file in S3, and the URI of the same is shipped to the indexing module that runs on an Ec2 instance. This indexing module fetches the dataset from S3 and constructs an index which it stores in SimpleDB. Here, three indexes i.e. SPO, POS and OSP are built to answer queries.

H2RDF [86] uses three indices i.e. SP_O, OS_P and PO_S that are stored in the form of key-value pairs into HBase tables. Here, SP_ indicates that a B+ tree is kept on the basis of combination of predicate and subject values. The SP_O index is responsible for the queries having either a bound subject and predicate or only a bound subject. The authors state that out of the six indices these three indices are sufficient to obtain an optimal performance. It is observed that all the eight possible triple patterns can be answered using just these three indices.

The performance of chain pattern queries is improved in RDFChain [22] by using a T_{com} table alongwith the SPO (subject-predicate-object) and OPS (object-predicate-subject) tables. The triples whose variables coexist in both object and subject parts of RDF triples are held in the T_{com} table and such triples are removed from the SPO and OPS tables. For processing the star pattern queries both T_{com} and SPO/OPS tables have to be accessed.

SHOE [66] produces atleast three partition views; S, OS and PO. It uses the partition information maintained in the metastore for query execution.

Rainbow [39] uses a hybrid indexing scheme for effective storing and query processing of Big RDF data. It creates a three table index i.e. SPO, OSP and POS to support any type of triple patterns. Here, the commonly used RDF data indices are stored in hash tables in a distributed storage for fast and random access.

The three permutations of Subject(S), Predicate(P) and Object(O) values in RDF triples that are materialized for creating index in SparkRDF [110] are SPO, POS and OSP. These three index tables are stored in HBase to achieve the desired index scan and search capabilities. The index schema used by SparkRDF has some advantages eg. efficient handling of multivalued properties. The import process requires two MapReduce jobs for each of these three index tables during bulk loading.

In Rya [91], RDF triples are indexed across three separate tables i.e. SPO, OSP and POS for satisfying all possible permutations of the triple patterns. The RDF triple is stored in Accumulo row ID in these tables and it differently orders subject, predicate, object for each table. For example, the SPO table stores a RDF triple in the row ID as (Subject, Predicate, Object), the POS table as (Predicate, Object, Subject), and the OSP table as (Object, Subject, Predicate). Thus in this approach, the RDF triples are stored and queried across multiple Accumulo tables by efficiently utilizing the row sorting scheme of Accumulo.

In AdHash [43], the authors build indices based on the conclusion that most of the search operations require a known predicate. Thus, in AdHash the triples are hashed in each worker by their predicate. Here, three indexes are built i.e. P, PS and PO. The search on predicate is immediately supported by Predicate index (P-index). The other two hashmaps which support searches with known predicate are Predicate-Subject (PS-index) and Predicate-Object (PO-index) indexes. The bucket of triples having same predicate are repartitioned using these two hashmaps based on their subject and object respectively.

The authors in [92] devised three types of indexes i.e. Constant-Constant Join (CCJ), Constant-Join-Free (CJF) and Join-Free-Free (JFF) indexes based on the typical join patterns. The Join-Free-Free (JFF) index is used for patterns with one join and two free variables, CCJ index used for patterns with two constants and one join variable and; CJF index for patterns with one constant, one join and one free variable.

Three indexes are built in HadoopRDF [29] i.e. spoc, posc and cosp. A path index based on HBase is proposed in [65] that reduces the number of joins as it is

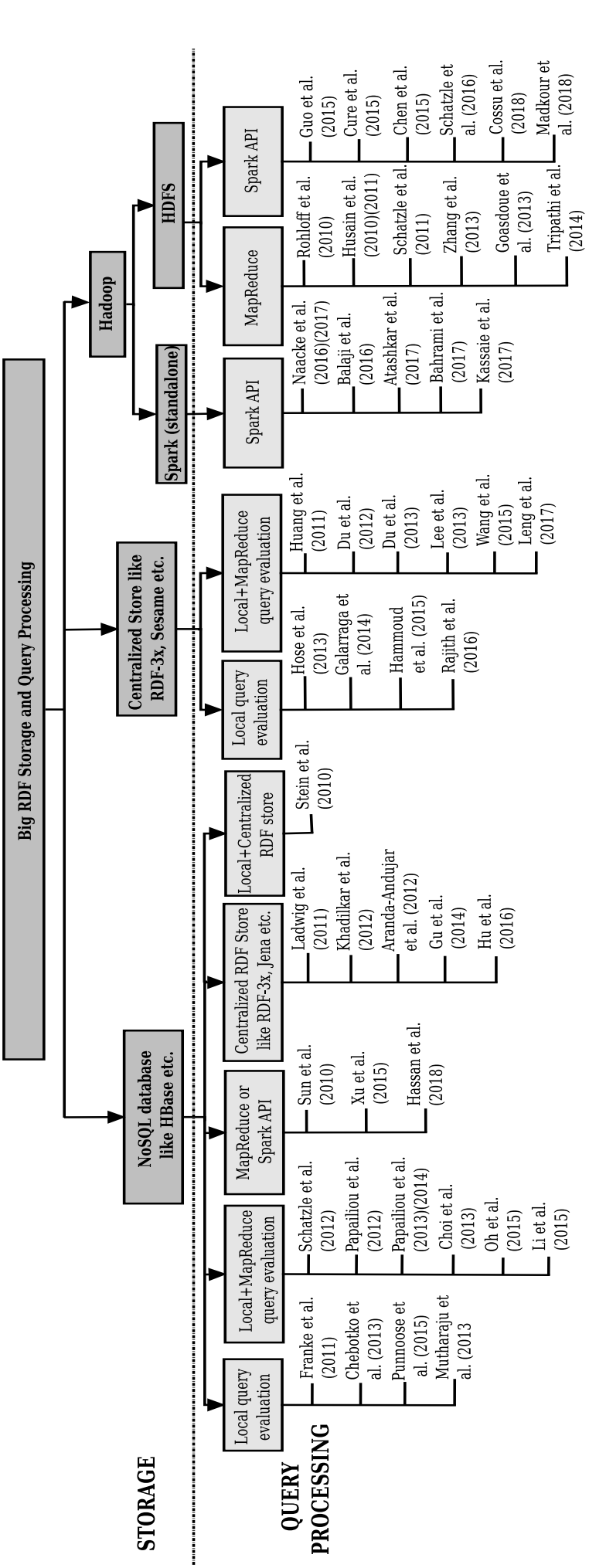

Fig. 6. A Classification of Storage and Query Processing Tools used in Big RDF Frameworks

more efficient for complex queries. Apart, from this two tables are created for each predicate i.e. PSO and POS. The number of joins can only be reduced but not completely eliminated by using the path index.

4.2.6. Two-index scheme

In [33], the authors used a storage schema where the RDF triples are stored in two tables i.e. T_{sp} (subjects of triples stored as row keys, predicates as column names and object as cell values) and T_{op} (triple subjects stored as cell values, predicates as column names and objects as row keys). The storage schema in the proposed framework requires that the RDF data be stored two times (replication factor is two) such that to ensure the robustness of the system.

The storage schema of MAPSIN [96] in HBase uses only two tables i.e. S_PO (indexed by subject) and O_PS (indexed by object). This schema is extended with a mapping of triple patterns that takes advantage of predicate push down filters in HBase to overcome any potential performance drawbacks that may arise because of the two table schema. The HBase Filter API is used to specify any extra column filters for the lookups of index tables. The predicate pushdown means that these filters are used at server side so, no irrelevant data is transferred over the network. The number of MapReduce iterations and HBase requests are reduce by using multiway join optimization in MAPSIN.

D-SPARQ [75] creates compound indexes that involves both subject-predicate and predicate-object pairs. Such type of index in MongoDB handles the queries on any prefixes of the index. Thus, the two indexes created in this framework are SP_O and PO_S.

4.2.7. One-index scheme

A simple index schema is used in Stratustore [100] i.e. SPO as it uses an entity oriented mapping where the data known about one subject; S is represented by one item. So, a single item for the entity “S" will contain data from every triple that contains S as the subject. Thus, one predicate defined for this subject will be represented by the other attributes and the objects will be represented by attribute values.

Oh et al. [81] proposed a framework which uses a single unified table called ‘Tuni’ containing both the encoded O_PS and S_PO patterns. For a S_PO pattern the identifier is “SO" and the encoded predicate and subject are each assigned to the column name and the row key respectively. Here, the column value has the value of an encoded object. Similarly, for the O_PS pattern the identifier is “OS", the encoded predicate and object are each assigned to column name and row key respectively while the encoded subject represents the column value.

The Classes Correlations with Property (CCP) index where the query input file is indexed by taking the data file names as an index in RQCCP [40]. The pattern matching is used in combination with this CCP index for obtaining the intermediate results for triple patterns.

Discussion

We observed that most Big RDF Frameworks use a Three-table index scheme. Also, most of the frameworks using this scheme store the indexes in a NoSQL database like HBase [39][86][9] and Accumulo [91]. The frameworks utilize the indexing and row sorting scheme of these NoSQL databases for efficient storage and querying across multiple tables. Although there are six probable permutations of the triple elements i.e. subject, predicate and object. The three permutations i.e. SPO, POS, and OSP are sufficient and necessary for efficient answering every possible triple pattern in a SPARQL query by only using a range scan. Most systems adopt this index scheme because of its compactness and generality.

5. Big RDF Retrieval

We also examined and classified the Big RDF frameworks according to common storage and query processing tools they use as shown in Figure 6. Most of the above discussed Big RDF frameworks fall into four broad categories discussed below:

- As shown in Figure 7, the first type of frameworks evaluate a query; Q by intra-partition processing i.e. locally in parallel with no inter-node communication in Case I. In Case II, where inter-node communication is required, the query needs to be decomposed into subqueries; q such that each subquery can be evaluated by intra-partition processing. In both cases, the local matching results for Q or q can be merged at the master (as shown in Figure 8) or a single slave node (as shown in Figure 7) to generate the final results of Q. If they are merged at a slave node, the slave returns the results to master from where the user can obtain results.
- The second category of frameworks evaluate query; Q locally by intra-partition processing in Case I, similar to the first type of frameworks. But in the

situations where inter-node communication is required, the query; Q needs to be decomposed into subqueries q. During inter-partition processing in Case II, the subqueries are evaluated locally and the MapReduce computation model is used for exchanging data between nodes in cluster. The MapReduce framework is also used for joining the intermediate results; as illustrated in Figure 8. In Case II, the final query results can be retrieved by master from HDFS where they are stored after joining.

- The third category of Big RDF frameworks, use MapReduce or Spark API for query processing. These frameworks may store Big RDF data in a NoSQL database or HDFS or Spark(standalone) in a cluster.
- Finally, the fourth category of frameworks use the Centralized RDF store like Jena, RDF-3x, Sesame etc. for SPARQL query processing and store Big RDF data in a NoSQL database.

5.1. SPARQL Query Processing

Agathangelos et al. [5] categorized some Big RDF frameworks using Spark for query processing based on the data model and processing of RDF data. The triple model based systems use the RDD API or Spark SQL abstraction for query processing. While the graph model based systems use the GraphX or GraphFrames abstraction for query processing.

Jena-HBase [60] provides support for SPARQL query processing, reification and inference by using appropriate Jena interfaces. This framework uses HBase instead of HDFS for data storage as HBase doesn't require MapReduce processing framework for accessing the data. Here, a corresponding layout which matches a given triple pattern is queried for the triples. The SPARQL query is submitted by user in PrestoRDF [72] over CLI which is translated into SQL by a SPARQL to SQL converter i.e. RQ2SQL that is custom made. This SQL query is then submitted to Facebook Presto. The Hive connector and Hive Thrift Server in Presto is used by PrestoRDF for running the generated SQL query on Big RDF data stored in HDFS and the results are returned back to the CLI.

5.1.1. Index lookups

The Sesame query processor is used for evaluating SPARQL queries in CumulusRDF [61]. This system only supports single triple pattern queries. The SPARQL queries are translated to index lookups of Cassandra indices by the Sesame processor. The filter operations and joins are processed by this processor on a allocated query node.

The query execution is performed in AMADA [9] on virtual machines within EC2. The submitted query in the system is directed to module for query processing running on an Ec2 instance. A lookup to the index in SimpleDB is performed by this query processor module for finding relevant datasets to answer query and the query is evaluated against these datasets. After this, the results are recorded in a file saved in S3 and the URI of this file is forwarded to the user for retrieving query answers. It is observed that SimpleDB only favors single relations queries (i.e. with no joins). In AMADA, the SPARQL queries are executed using RDF-3x.

In [65] different means of querying are proposed as it requires to only lookup the path index and some particular tables for processing simple SPARQL queries while cascading MapReduce is used to process complex queries. This framework also supports regular expression filter and range queries.

5.1.2. Adaptive query execution

In SHAPE [62], the distributed query processing component consists of three main tasks: query analysis, query decomposition and generating distributed query execution plans. The query analyzer determines whether or not a query; Q can be evaluated using intra-partition processing as can be seen from Figure 8. SHAPE also evaluates queries locally and uses MapReduce for joining intermediate results. All queries that can be evaluated by intra-partition processing will be sent to the distributed query plan execution module. For those queries that require inter-partition processing, it invokes the query decomposer to split the query into a set of subqueries, each can be evaluated by intrapartition processing. The distributed query execution planner will coordinate the joining of intermediate results from executions of subqueries for producing final query results. SHAPE uses no query optimization and this is left for future work.

5.1.3. Query decomposition

In HAQWA [26], the queries are decomposed into a set of sub-queries for local evaluation. Among these subqueries each of them is a prospect for being the starting point or the seed query for the evaluation of query pattern. In the second step, data encoding is performed after the data is allocated to the nodes in cluster. Some

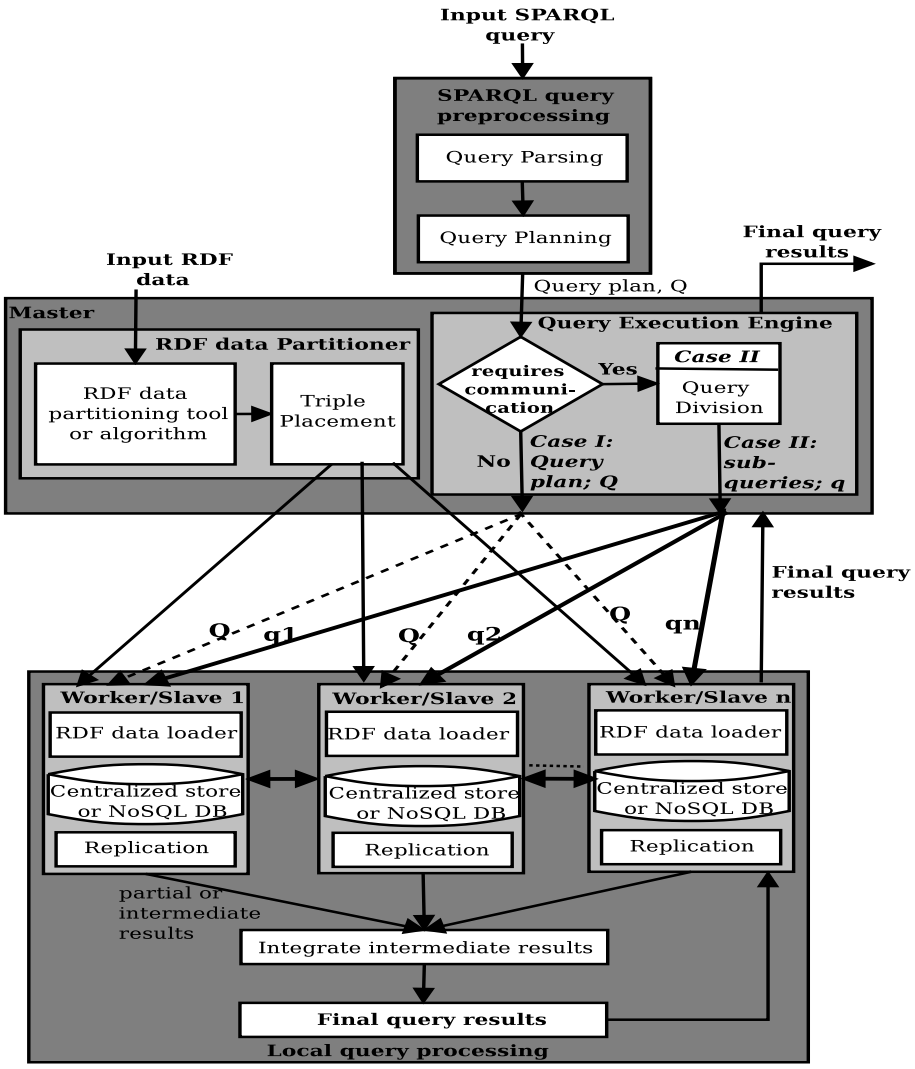

Fig. 7. A Big RDF Framework using Centralized store or NoSQL database for storage and evaluate SPARQL queries/sub-queries locally **Case I:** Intra-partition processing (no inter-node communication); **Case II:** Inter-partition processing (inter-node communication)

of the advantages of this step include reduction in data volume as subject, predicate and object of triples are stored as integers and not as strings. In the third step, the query received by HAQWA is converted into Spark program and the Spark API is used for query processing.

5.1.4. Basic Graph Pattern (BGP) matching

S2X [95] uses graph specific optimizations in such a way that graph-parallel and data-parallel computation are combined resulting in minimal data movement. Here, a vertex centric algorithm is used for BGP matching. These vertex centric algorithms are used such that each vertex is able to send, receive and process messages between its neighborhood vertices.

5.1.5. Subgraph matching

The AggregateMessages operator of GraphX used for subgraph matching in [59] provides two functions i.e. mergeMsg and sendMsg. The sendMsg can be regarded as a map function using which the current triple of BGP is matched with all the triples of graph. In case, a match is found then different messages will be prepared and it sends messages by the sendMsg function to the destination and source of the triple. After that, the function of mergeMsg will be used as a reduce function

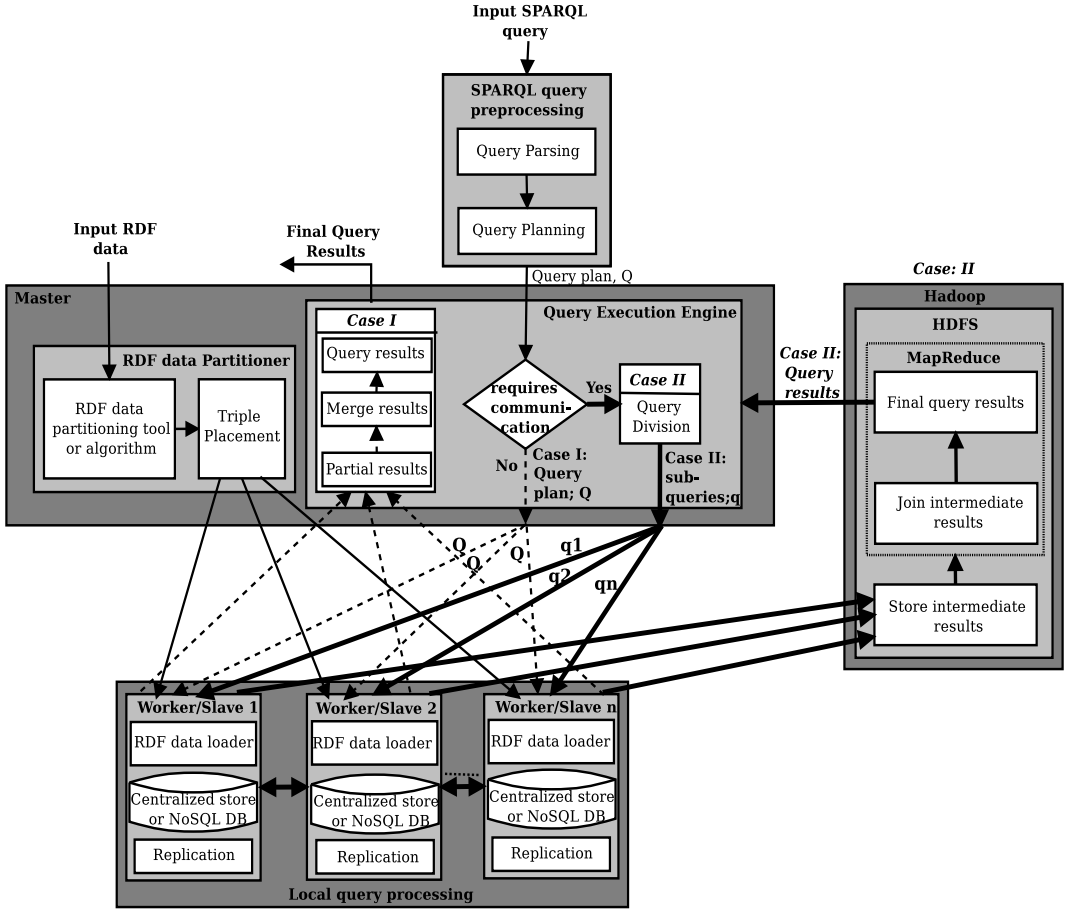

Fig. 8. A Big RDF Framework using Centralized store or NoSQL database for storage and Local+MapReduce scheme for query processing. **Case I:** Intra-partition processing (no inter-node communication); **Case II:** Inter-partition processing (inter-node communication)

using which the messages that are received will be combined at their destination vertex. Finally, after all the BGP triples are evaluated, the final M_T tables of end vertices and which hold the partial results are joined for generating the query answer.

5.1.6. Other

SHARD [93] uses the MapReduce implementation for SPARQL query processing. In the first map of MapReduce step, the triple data is mapped to a list of variable bindings that satisfy the first clause of query. The list of variable bindings is the key of the map step. The duplicate results are removed in the reduce step and they are saved to disk using variable bindings as the key. So, it uses a clause-iteration technique for SPARQL query evaluation. This framework doesn't perform any query optimization for improving the query performance.

In [49] the SPARQL queries are translated into Spark SQL by the query compiler based on data storage scheme in the two systems i.e. HBase and Cassandra. Here, for processing Big RDF data in Spark API a single DataFrame is created for a HBase table and the table is cached in-memory. From HBase table a DataFrame is constructed for each subject and from that DataFrame temporary views are created. Similarly, in Cassandra it creates a DataFrame for each property that is cached in memory.

5.2. SPARQL Query Optimization & Processing

5.2.1. Query decomposition

In [20], the query is partitioned into individual jobs and each of them must be further divided into several independent subtasks. Then, each of these jobs can be

executed using MapReduce. The MapReduce jobs that are generated from the JobPartitioner will be executed one by one and the succeeding job may use the preceding jobâĂŹs output as its input. The authors used the optimization technique that independent Map-Reduce jobs can be parallelly processed.

In [28], the SPARQL queries are firstly decomposed into many individual star join queries. These star-join queries are locally processed by nodes efficiently with no communication and the query results can be locally saved for later chain-join processing. The star-join queries are analyzed by the query optimizer in this framework for generating an optimized join order. Here, each node is equipped with a RDF-3x engine and the intermediate results are saved in HDFS. After a series of chain-join operations it generates the final query results. This framework guarantees efficient local processing of star-join queries as triples of same entity are allocated to the same segment.

HadoopRDF [29] executes SPARQL queries as a series of MapReduce jobs. Also, SPARQL queries are decomposed according to the partitioning scheme so each sub-query can find the related nodes easily. The MapReduce framework is used for data processing and to arrange the execution steps for queries. After the SPARQL queries are executed in the triple-store, MapReduce jobs are used to merge the SPARQL queries execution results based on the relation between them.

The authors in [107] use a lightweight sideways information passing (SIP) technique for passing join information across MapReduce jobs and to reduce the number of intermediate results in join operations. The SPARQL queries are decomposed based on a hybrid partitioning strategy that is schema-based and a left deep tree is established for obtaining an ideal order of execution by using some selectivity estimation strategies. This framework translates thus obtained subqueries into SQL queries and they are dispatched to underlying RDBMS i.e. HadoopDB for evaluation. These subqueries are processed in an ordered manner means one job for a subquery and are then joined in a left deep tree manner i.e. one job for a join.

Sempala [98] consists of a query compiler to translate SPARQL queries into SQL of Impala on the basis of proposed data layout. Also, it can easily answer the star patterns without requiring any join. Jena ARQ is used here to parse the SPARQL query into equivalent algebra tree. Some of the simple algebraic optimizations used in Sempala are Filter pushing etc. For processing SPARQL queries, ProST [24] splits up the query into several sub-parts and each subquery is executed using the most suitable storage approach. Then, next the results from these subqueries are merged together to obtain the final result.

The different parts of query are run in parallel in D-SPARQ [75] by identifying following patterns in the query i.e. independent triple patterns that can be run in parallel, star pattern (subject-subject join) and pipeline or chain pattern (subject-object, object-object, object-subject joins). It maintains information for each predicate in the dataset and uses this information for reordering triple patterns in the star-pattern queries. After identifying the pattern in the input query the query processing is done by MongoDB which uses the appropriate indices while retrieving records from database.

5.2.2. Adaptive query execution

Huang et al. [52] proposed a framework where it is estimated that if a query can be answered completely in RDF-3x without any shuffling of data in Hadoop. Such query is called a PWOC (parallelizable without communication) query. If the query is not PWOC then it automatically decomposes the queries into parallelizable chunks or PWOC subqueries that can be performed individually with zero communication within partitions and the MapReduce framework of Hadoop is used for combining the resultant distributed chunks as shown in Figure 8. This system uses a heuristic to bunch as many joins as feasible in the same job thus, generating query plans with limited MapReduce jobs. Here, execution of queries is done in RDF-3x and/or in Hadoop, also, here the maximum possible processing is pushed into RDF-3x and the remaining is left for Hadoop.

In AdHash [43], queries can be evaluated in distributed or in parallel mode, the distributed mode is for queries that require communication and parallel mode is for queries which can be resolved with no communication. The queries evaluated in the parallel mode are autonomously scheduled by the workers. The query planner at the master consults the redistribution controller for deciding whether to execute a query in the parallel mode. If the query is to be executed in distributed mode a global query plan is devised by the locality-aware planner. If a query can be answered in the parallel mode, the master broadcasts it to all workers.

The query execution engine in H2RDF [86] uses a join cost model that chooses the best join algorithm

and its mode of execution i.e. centralized/distributed. H2RDF outperforms some other distributed solutions for multi-join and non-selective queries while for selective queries it shows comparable performance to centralized solutions. For queries with high selectivity and small input centralized execution is more effective than distributed processing. H2RDF executes evaluates centralized joins in a single cluster mode while distributed joins are executed using MapReduce. In centralized mode, the SPARQL query is evaluated at a single node in the cluster.

H2RDF+ [85][87] uses several optimizations such as intermediate result materialization. It uses lazy materialization for minimizing the intermediate results size. Here, the order of joins and centralized/distributed scenario during query execution is decided by using a join cost model and a greedy planner. H2RDF+ processes only the amount of data necessary for each join and not the whole dataset. Similar to H2RDF, in H2RDF+ also the query execution can be performed in centralized or distributed manner as the greedy planner with the help of cost model (uses stored statistics) estimates cost in both scenarios and generates a plan to be executed either on a single node in cluster or in MapReduce.

The framework proposed in [105] uses the principle of selectivity estimation for evaluating predicates with low selectivities first such that the number of tuples that are involved in joins are reduced. To facilitate query processing in this framework, each element of RDF triples is encoded into integers before query execution. Here, SPARQL queries are rewritten to obtain the query patterns. The focus during query evaluation is on finding the vertex of division for each query pattern. In case, where multiple queries share a predicate then object or subject corresponding to this predicate is chosen as the vertex of division for avoiding redundancy. To improve query evaluation efficiency the one which produces minimum number of intermediate results must be chosen.

JARS [92] uses a join pattern analysis that is rule-based and it performs execution using concurrent bushy joins on all the nodes in the initial step. The algorithm for processing queries checks the number of triple patterns in the BGP of SPARQL query and the number of join variables for checking if the query can be computed with no inter-node communication. The order of processing query is estimated by the query planner on the basis of number of cardinality estimates, join variables and constants. The simple and star-pattern SPARQL queries are locally executed and the query execution performance is improved by efficient co-location of joinable triples. It decomposes the composite queries and executes them in parallel.

The system proposed in [50] processes complex SPARQL queries efficiently by exploiting workload information and works on a similar approach as proposed by Huang et al. The queries containing single triple patterns or star queries are termed as one-pass queries and, it evaluates them in parallel manner on each partition. The complex queries or multi-pass queries are split up into multiple one pass subqueries, evaluated in parallel and the results are merged at the coordinator. Here, instead of using the expensive MapReduce jobs for computing joins connecting the subqueries, pipelining at the coordinator and efficient merge joins are used as an alternative that results in less overhead.

In DiploCloud [108], data that is partitioned and its corresponding local indices are held by the workers in the system. The three main data structures stored by workers are, a type index, a molecule index and local molecule clusters. The subqueries are run by the worker nodes which transmits the results to the master. The queries which consist of one BGP such as star-pattern queries are evaluated in a parallel manner and with no central coordination. DiploCloud chooses one of the two execution strategies for distributed joins, (i) when intermediate result set is small in size, everything is shipped by DiploCloud to the master which implements the join; (ii) if the size of intermediate result set is large, then a distributed hash-join is performed by DiploCloud.

In DREAM [42], the master machine involves a query planner. The SPARQL query is submitted by the client to the master machine for transforming it into a graph and feeding this graph to the query planner. This graph is partitioned into sub-graphs by the query planner subsequently, this subgraph is placed at a single slave machine by the master and all the machines execute it in parallel. Finally, during query execution the slave machines exchange intermediate auxiliary data, join the intermediate data and generate the final query results. The communication cost is negligible in DREAM as it shuffles no intermediate data and only exchanges minimal metadata and control messages. DREAM uses an adaptive approach as the queries are automatically run on various number of machines depending upon their complexity.

5.2.3. Basic Graph Pattern (BGP) matching

The framework proposed in [33] performs query optimization by triple pattern reordering. For executing SPARQL queries over the proposed storage schema in the MySQL cluster, a query translation algorithm for basic graph pattern (BGP) that translates SPARQL queries to SQL and generates flat SQL queries is presented by the authors. This flattening of queries in MySQL cluster eliminates the overhead of query optimization by triple pattern reordering as a relational query optimizer is efficient enough to select a good query plan automatically.

Chebotko et al. [18] uses a novel SPARQL evaluation technique that heavily relies on indices for computing the costly join operations and representation of triple positions in query by numeric values rather than the actual triples. Thus, eliminating the intermediate data transferred in the network. This framework performs expensive join operations for query processing by utilizing compact bitmap indices and the RDF graph needs to be accessed only one time for replacing positions of triples in the SPARQL query with actual triples. Here, query optimization is done by triple pattern reordering and the criteria used for reordering triple patterns is the same as used in [33].

Hassan et al. [48] proposed a framework where Apache Spark and Drill computation models are used for query processing. The authors also propose a compiler for query validation and building query parse tree. The SPARQL query is translated into SQL statement using this compiler. The compiler also optimizes SPARQL queries by triple pattern rewriting and triple pattern reordering (for avoiding further cross joins). The tables are chosen from the predicate position of each triple pattern in VP and VPExp approaches. While, a combination of two triple patterns with any existent correlation is used for choosing tables in 3CStore. A BGP is taken as input here in each algorithm for generating the lost of conditions and projections, the table names are used to produce subqueries for the triple patterns.

5.2.4. Subgraph matching

A RDSG-based iterative model is used in [19] for processing SPARQL queries by implementation of iterative join operations with distributed memory which helps in evading the high costs for intermediate results. In this framework, a new cost model is proposed based on the query and data model of RDF. SparkRDF realizes the SPARQL query processing problem by transforming the query to the problem of joining and iterative subgraph matching. Thus, here the final resultant subgraph is obtained by computing matchings for every triple pattern in SPARQL query and performing a subgraph join operation on these matchings.

The second module i.e. Query Graph Generation of the framework proposed in [13] analyzes the SPARQL queries for identifying the triples to generate query graph. This query graph is examined in the input RDF graph to answer a SPARQL query. Query optimization occurs in the third module where frequency count of each distinct predicate (or edge) in the input RDF graph is computed. In the fourth module i.e. Local Search Space Pruning, the irrelevant triples are eliminated from the data graph with respect to the given query using a table lookup mechanism and thus reducing the search space. Finally, the fifth module i.e. Query Processing accepts as input the optimized triple patterns and RDF triples that are locally pruned to perform subgraph matching and identifying query relevant triples.

5.2.5. Graph exploration

In Trinity.RDF [112], the query is submitted by the user to a proxy. The query plan is generated by this proxy and submitted to all Trinity machines holding RDF data. Each of these machines under the coordination of proxy execute this query plan. When the binding of the query variables are calculated, all the trinity machines return these bindings or query answers to the proxy which assembles the final results and sends them back to the user. This system adopts a cost-based approach that finds an optimal exploration plan on the basis of some heuristics. Here, dynamic programming is used for exploration optimization. Trinity.RDF leverages efficient graph exploration for query processing and thus, greatly reduces the volume of intermediate results.

TriAD [41] utilizes a bottom-up dynamic programming algorithm for join order enumeration. This system uses both asynchronous inter-node communication and intra-node multi-threading for running different join operators of a query plan in a parallel and distributed manner. TriAD facilitates join-ahead pruning in a distributed environment by using a novel form of RDF graph summarization (summarizing large RDF graph into a smaller graph). Here, the query optimization is employed at two stages i.e. over both the RDF data and RDF summary graphs. It executes a query plan in a multi-threaded fashion at each slave.

5.2.6. Parallel Triple pattern matching and execution

In Stratustore [100], SPARQL query processing is done using the Jena Graph API. For query answering the SPARQL query strings are divided into parts which are then mapped into much simpler SELECT queries. SimpleDB evaluates these subqueries separately, merges the results of these subqueries and returns final results to the user. The set of triple patterns constructed from the query are handed to Stratustore via Graph API in the second step. Next, it groups triple patterns by the subject, creates a SELECT query for each group of triple patterns and poses them in parallel to SimpleDB. SimpleDB returns the list of items matching to these SELECT queries next, Stratustore reconstructs the triples based on these items and performs necessary joins. Finally, Jena framework applies any required additional processing and returns the result to user.

Rainbow [39] uses Sesame framework for query processing. The SPARQL queries are executed with Sesame using Redis as the backend for RDF storage. This framework uses triple pattern reordering as the query optimization approach according to which high selectivity triple patterns are executed prior to lower selectivity ones. A metadata table is located in the client's memory in Rainbow, the table is updated when new RDF data is added. The metadata in this table is useful for estimating the selectivity of triple patterns and query performance. Here, query execution is done by triple pattern matching and each triple pattern match execution requires to find an appropriate storage layer as per its type.

ScalaRDF [51] is implemented based on Redis and Sesame for the data query engine. The SPARQL query is parsed by the master and it also creates an optimal query plan. The slaves are responsible for the storage of triples and execution of the query plans. Monitor component manages the task of storage and query execution, it reports the results to the master. ScalaRDF uses the cardinality generated during data loading step to optimize queries and for generating an optimized query model afterwards. The architecture and the query processing workflow of ScalaRDF is quite similar to that of Rainbow [39]. The triple pattern matching process is accelerated by performing the matching operation in parallel.

Rya [91] utilizes the OpenRDF Sesame SAIL API for storing and querying RDF triples from Accumulo. Rya executes SPARQL queries by using index nested loop join. Rya utilizes Hadoop MapReduce to run large batch processing jobs on the dataset. Triple Pattern Reordering approach is used in Rya for query optimization and for the same query selectivity is estimated using MapReduce jobs in Hadoop. The statistics thus generated are stored in a table so they can be retrieved later. The row sorting scheme of Accumulo is used for efficient storage and querying of triples across multiple Accumulo tables.

The logical optimizer in [103] performs the tasks of interpreting statements in Pig Latin and constructing an optimized logical plan. The order of operations may be changed, two operations may be combined into an equivalent operation or materialized data may be used for query performance improvement by the logical optimizer. The logical plan is compiled into multiple MapReduce jobs by the physical optimizer used here. The query optimization techniques used here are filter operations that are applied while loading RDF data so, only the requisite RDF data is read from HDFS. Some of the strategies used for optimizing multiple join operations include join reordering, reusing historic results, selecting faster join algorithms against specific data distributions etc.

5.2.7. Heuristics based processing

In [102], a SPARQL Basic Graph Pattern (BGP) processing strategy based on MapReduce is also provided according to the storage schema. A typical BGP is processed using multiple MapReduce jobs and a greedy method that eliminates the multiple triple patterns is used for selecting the join key. The proposed framework uses a greedy strategy to limit the number of MapReduce jobs that are required for complex SPARQL queries. For a query that requires no join there won't be any MapReduce job, in that case result of the only triple pattern will be directly used as the final output.

In Partout [34], the queries are issued by the user at the coordinator, which generates query plans suitable for distributed query execution. While, the actual data is situated at the hosts which are responsible for data partitions hosting. In the cluster, a part of query is executed by each host over its local data and the results obtained are sent to the coordinator. The coordinator finally holds these query results. The global query optimization algorithm utilized by Partout eliminates the requirement of a 2-step approach to start with a plan that is optimized relative to the selectivity of predicates in query. An efficient query plan is obtained according to the distributed scenario by applying heuristics. Also,

every host in the cluster depends on the RDF-3x query optimizer to optimize its local query plan.

SparkRDF [110] uses some heuristics to determine the order of joins in SPARQL queries. The join planner implemented in SparkRDF for query optimization generates an optimal order of joins on the basis of triple patterns and query algebra. Then, SparkRDF pipe-lines the iterative joins in Spark RDD and gives the final result as output. Thus, the intermediate join result is directly pipelined to the next join iteration in main memory thereby eliminating the irrelevant disk I/O. The queries with no shared variables will not require any join thus, there will be no Spark jobs in such case. The result of the triple patterns can be obtained directly from the HBase tables which can be output directly as the final query result.

RQCCP [40] improves the SPARQL query performance over Big RDF data 3 aspects i.e. reduction of input data, disk I/O and network overhead. Several strategies are used in RQCCP for reducing query execution time. The index is applied for query optimization, to limit the input for query which thus filters the irrelevant data, intermediate results generated during query execution are cached in memory for reducing the disk and network I/O also, the intermediate results are joined by adopting a greedy policy.

The authors in [77][76] implemented five SPARQL query processing strategies i.e. SPARQL SQL, RDD, DF, Hybrid RDD and Hybrid DF on Apache Spark. The hybrid solution overcome the drawbacks SPARQL SQL, RDD and DF solution. This hybrid strategy combines the partitioned and the broadcast join. The knowledge of existing data partitioning scheme is exploited for combining broadcast joins with local partitioned join. The hybrid strategy implements a simple dynamic greedy strategy using a cost model for SPARQL query optimization. A best query plan that depends on number of machines is prepared based on the cost model. For query execution, a pair of subqueries and the join operator that generate minimal cost are chosen using the cost model.

The algorithm proposed in [54] uses backtracking in combination with a two coloring scheme for generating multiple plans. Then, a best plan is chosen by this algorithm based on a cost model. The number of jobs required for answering the query and the sequence as well as inputs of those jobs is determined by the plan. The cost estimation is done for processing a query as more than one job may be required for answering a query, so the cost of each job needs to be estimated.

The framework proposed by Husain et al. [53] uses a greedy approach for determining the minimum number of Hadoop jobs to be executed for solving a query. Here, summary statistics are used for estimating the join selectivity. A heuristic cost based algorithm is used for implementing query processing on Hadoop, to produce a query plan having minimal number of Hadoop jobs to join data files and input query evaluation. The authors use a rewriting algorithm that helps in reducing the input size and it may also eliminate a few joins. Husain et al. uses Hadoop MapReduce processing framework to answer the queries.

5.2.8. Triple pattern mapping

The two join techniques in MapReduce are Reduce side or Repartition join and Map-side join. MAPSIN [96] overcomes the drawbacks of both these techniques by transferring only necessary data over the network and by using the distributed index of HBase. The join between two triple patterns is computed in a single map phase by using the MAPSIN join technique. In comparison to the reduce-side join approach which transfers lot of data over the network, in the MAPSIN join approach only the data that is really required is transferred. The resultant set of mappings computed finally are stored in HDFS. MAPSIN also supports multiway join optimization. This kind of optimization is efficient for queries which share the same join variable such as star-pattern queries.

RDFChain [22] decreases the number of map jobs required in multiway joins. It estimates the cost of processing joins using statistics to split the query and the queries separated includes as many triple patterns as possible that can be processed in a map job. In RDFChain, query planning is done by deriving a logical plan for the SPARQL query graph. This logical plan consists of a set of triple pattern groups (TPG's) and their join order. During query execution, the logical plan is transformed into a physical plan where each TPG in the logical plan is transformed into a map job in the physical plan. For star pattern join queries, RDFChain requires to only retrieve a single row through a single storage access in a map job; for chain pattern join queries only T_{com} table needs to be scanned that reduces the number of storage access.

The framework proposed by Oh et al. [81] overcomes drawbacks of both MAPSIN [96] which is optimized for only star-shaped and not chain-shaped join queries;

and RDFChain [22] which optimizes process for chain-shaped queries but requires to access two tables for star-shaped queries. In this framework, only one HBase table needs to be accessed for both chain and star shaped queries. Here, the RDF data is input to the map phase so no reordering is required for query evaluation and no shuffle and sort phases are required for star and chain shaped queries. The abstract RDF data is utilized for finding out the partition where the result lies and thus, the amount of input to MapReduce jobs is reduced.

The Jena ARQ engine is used in [97] for checking syntax and generating algebra tree. The optimization of SPARQL queries based on Pig Latin means reducing the I/O required for transferring data between mappers and reducers as well as the data that is read from and stored into HDFS. Some of the query optimization strategies used by PigSPARQL are the early execution of filters, selectivity-based rearrangement of triple patterns etc. A fixed scheme that uses no statistical information on the RDF dataset i.e. Variable Counting is used by PigSPARQL. The resultant Pig Latin script is automatically mapped onto a sequence of Hadoop MapReduce jobs by Pig for query execution.

The MapReduce framework in this architecture [8] has three subcomponents i.e. query rewriter, query plan generator and plan executor. First, the SPARQL query taken as input from the user is fed to the query rewriter and query plan generator. Then, this module picks up the input files for deciding the number of required MapReduce jobs and then it passes this data to Plan executor module that uses the MapReduce framework for running these jobs. Finally, it ships the query result from Hadoop to the user. The data stored to a list of variable bindings is mapped by the initial map step for satisfying the first query clause. After this is done, the duplicate results are discarded by the reduce step and it uses the variable binding as key for saving them to the disk.

SPARQLGX [38] directly compiles the SPARQL queries into Spark operations. For query optimization, SPARQLGX uses its own statistics. Also, there is an additional feature in SPARQLGX named as SDE for direct evaluation of SPARQL queries over Big RDF data without any extensive preprocessing. This feature is valuable in cases of dynamic data or where only a single query needs to be evaluated. In SDE, only the storage model is modified so instead of the predicate files directly the original triple file is searched for query evaluation and the rest of the translation process remains same. This framework maps the triple patterns in SPARQL queries one by one to Spark RDD.

In S2RDF [99], the query evaluation is based on Spark SQL, which is the relational interface for Spark. The SPARQL query is parsed into a corresponding algebra tree using Jena ARQ. The equivalent Spark SQL expression is generated based on the ExtVP schema by traversing the tree from bottom up. The equivalent Spark SQL query generated after mapping is executed by Spark. S2RDF optimizes queries using the technique of triple reordering by selectivity estimation. For evaluating the generated SQL query the precomputed semi-join tables can be used by S2RDF if they exist, or it alternatively uses the base encoding tables.

5.2.9. Other

In EAGRE [113], the queries are not executed until the data blocks which contain the query answer are determined. For SPARQL query evaluation, firstly the entity classes are identified by using an in-memory index for compressed RDF entity graph on a query engine. After this, the query is submitted to the worker nodes maintaining the RDF data where the query coordinator at each worker node participates in a voting process that decides the scheduling function of the distributed I/O operations. EAGRE reduces the cost of disk scans and the total query evaluation time by using a distributed I/O scheduling mechanism. After the workers complete their local I/O operation they use the scheduler for feeding workers with the gathered statistical information of processed data.

Some of the other components of SHOE [66] include compiler, optimizer, partitions metastore and an execution engine. This framework uses Map-Reduce for query execution that evaluates query with a sequence of pair-wise joins on the filtered triples. Also, a partition centric identification join strategy is used which mainly focuses on taking advantage of the partition-wise joins. The query plans are generated by using some optimization steps such as statically binding partition keys to leave nodes, applying greedy matching strategy to whole attributes permutation, star-joins merging plus the statistics-driven partition join, and dynamic partition binding.

For SPARQL query processing, CliqueSquare [35] relies on a clique-based algorithm, which produces query plans that minimize the number of MapReduce stages. The algorithm is based on the variable graph of a query and its decomposition into clique subgraphs. This algorithm works in an iterative way to identify cliques

and to collapse them by evaluating the joins on the common variables of each clique. The process ends when the variable graph consists of only one node. Since triples related to a particular resource are co-located on one node CliqueSquare can perform all first-level joins in RDF queries (SS, SP, SO, PP, PS, PO, etc.) locally on each node and reduce the data transfer through the network. It allows queries composed of 1-hop graph patterns to be processed in a single MapReduce job.

The CSQ algorithm proposed by Goasdoué et al. in [36] firstly, builds the query plan, the query variable graph is decomposed into several cliques. The clique is defined as a set of variable graph nodes that are connected with edges having a certain label. From the perspective of query optimization, clique decomposition means to identify the partial results which need to be joined. The next step is clique reduction which is done based on clique decomposition. In clique reduction, the joins identified by decomposition are applied. The cost of the query plan is defined as an estimation of the total work required by the MapReduce framework.

The authors in [104] observed that if the clauses that generate less data are executed first then better response time is obtained as less data means less effort in joining data generated by the 2 clauses. Thus, the cost of join operation can be reduced by reordering the queries such that the clauses generating less data are at the top. This framework uses the SHARD [93] triplestore to persist RDF triples thus, it also uses the clause iteration approach of SHARD to run SPARQL queries over Big RDF data.

In [114] the SPARQL queries are directly mapped to a sequence of MapReduce jobs that may employ some hybrid join strategies (a combination of map-side, reduce-side and memory backed join). The problem of selecting the MapReduce jobs and their ordering is solved by using a novel tree structure i.e. All Possible Join (APJ) tree that implies all the possible joins plans which are to be examined. Some of the query optimization techniques used to improve performance and join processing are hybrid join and bloom filter.

SPARTI [71] initially it analyzes the frequent join patterns in the query workload. So, initially SPARTI behaves like any other system using VP scheme and reads the entire partitions to answer SPARQL queries. This framework firstly, identifies the properties and join patterns in the queries and every property as well as query join pattern is matched with a SemVP partition and Semantic filter respectively. In the case where a match is found, the semantic filter is read for answering a query instead of reading the entire partition for a property. It is similar to S2RDF [99] in the respect of employing reductions for achieving better query performance.

[14] proposed a new querying model which eliminates the driver based aggregations and data shuffling from joins. The bi-directional structural filtering mechanism is followed to prune the path at each level. A modified adjacency list format is used here to model data where neighborhood of a vertex is stored by sorting it in the order of edge labels. A single scan is made across the graph to retrieve set of likely candidates for each query segment and these vertices are called the 'prune set'. This prune set is generated at the start of query execution.

In Spar(k)ql [37], the process of query answering by vertex programs is implemented during the second step. Here, every nodes gets messages from its neighbors and then the sub-results are calculated based on the information stored and these incoming messages. In third step, the algorithm in GraphX uses iterations for analyzing the graph and during each iteration it checks all the active edges (an edge is called active if one of its nodes is active). The fourth step comprises of formulating query plan for evaluating SPARQL queries. Finally, queries are processed, by traversing the query plan in a bottom-up manner where it iterates over the edges for each node to find the corresponding matches.

Discussion

We observed that many of the Big RDF frameworks use a adaptive query execution strategy for processing SPARQL queries. With this strategy the frameworks can decide on the centralized or distributed join execution for queries. In centralized or local mode, the SPARQL query pattern is searched completely by each worker in the cluster or by a single worker in the cluster such as in H2RDF and H2RDF+. The advantage with adaptive execution is that simple or single triple pattern queries can be processed with negligible communication cost and no overhead.

An overview of Big RDF frameworks storing Big RDF data in plain files is given in Table 7. The Big RDF frameworks using partitioning but no indexing are outlined in Table 8. In Table 10 we give an overview of the Big RDF frameworks using both partitioning and indexing for query performance improvement. Finally, the Big RDF frameworks using indexing and no partitioning are examined in Table 9.

Author	Storage Model and Tool	Partitioning	Indexing	Query Optimization	Query Processing Tool and Technique
Husain et al. (2010) [54]	Binary Table HDFS	—	—	Job cost estimation, Graph coloring for plan generation	MapReduce MapReduce job-based query processing
Tanimura et al. (2010) [103]	Binary Table HDFS	—	—	Data filtering, join reordering, reusing past results, logical optimizer	Pig+MapReduce Pig script translated to mapreduce job, mapreduce job-based processing
Huain et al. (2011) [53][55]	Binary Table HDFS	—	—	Job cost estimation, heuristic model, greedy approach	MapReduce Heuristic-based query processing
Schätzle et al. (2011) [97]	Binary Table HDFS	—	—	selectivity-based triple pattern reordering, Variable counting, early execution of Filters, multi-joins	Pig+MapReduce Reduce-side join based query execution
Ali et al. (2012) [8]	Binary Table HDFS	—	—	Caching intermediate results	MapReduce Mapreduce-join iterations
Zhang et al. (2012) [114]	Binary Table HDFS	—	—	All Possible Join tree (APJ-tree), hybrid join, bloom filter, cost model-based join processing	MapReduce APJ-tree based mapreduce job scheduling technique
Graux et al. (2016) [38]	Binary Table HDFS	—	—	Statistics-based join reordering, selectivity-based triple pattern reordering	Spark API Maps TPâĂŹs one by one to Spark RDD API
Schätzle et al. 2014 [98]	Property Table HDFS (Parquet)	—	—	Filter pushing, Join reordering	Impala SPARQL query translated to Impala SQL
Gombos et al. (2016) [37]	Graph-based Spark (standalone)+ GraphX	—	—	Triple pattern reordering, BFS algorithm to generate query plan, sub-results stored in tables at each node	GraphX API Node & Message model
Bahrami et al. (2017) [13]	Graph-based Spark (standalone)	—	—	Heuristics-based triple pattern reordering, Local search space pruning	GraphFrames API Subgraph matching
Hammoud et al. (2015) [42]	Any RDF-3x	—	—	novel graph-based, rule-oriented query planner, cost model, job scheduler (random & greedy)	RDF-3x Query Partitioning, Query Decomposition, Adaptive execution
Hassan et al. (2018) [48]	Binary Table HDFS (Parquet)	—	—	Triple pattern rewriting and reordering	Apache Zeppelin Spark SQL temporary views
Hassan et al. (2018) [49]	Mixed HBase+Cassandra	—	—	—	Spark SQL Distributed in-memory query processing
Schätzle et al. (2015) [95]	Graph-based GraphX	—	—	—	GraphX+Spark API Graph+Data Parallel query execution
Kassaie et al. (2017) [59]	Graph-based Spark (standalone)	—	—	—	GraphX API Subgraph matching
Cheng et al. (2012) [20]	Triple Table HDFS	—	—	—	MapReduce Uniprocessing, area optimization, multitasking

Table 7
A comparison of Big RDF Frameworks using storage & query processing techniques

Author	Storage Model and Tool	Partitioning	Indexing	Query Optimization	Query Processing Tool and Technique
Huang et al. (2011) [52]	Triple Table RDF-3x	Graph partitioning (using METIS)	—	Heuristic-based query decomposition	RDF-3x+MapReduce PWOC query in RDF-3x & rest in MapReduce
Du et al. (2013) [28]	Triple Table RDF-3x	Hash partitioning (on subject)+Query workload aware distribution	—	Join reordering, Caching intermediate results	RDF-3x+MapReduce Query Decomposition,BLQU-star join queries, chain join query processing
Goasdoué et al. (2013) [35]	Triple Table HDFS	Subject, Property & Object based Partitioning	—	Clique decomposition & reduction variable clique	MapReduce Clique-based algorithm
Zhang et al. (2013) [113]	Triple Table HDFS	Graph partitioning (using METIS)	—	Job scheduling, bloom filter, distributed I/O scheduling	MapReduce Distributed I/O scheduling, postponing of MapReduce jobs
Wu et al. (2012) [107]	Binary Table HadoopDB	Horizontal partitioning	—	Selectivity-based subquery reordering	MapReduce Light-weight sideways information passing
Hose et al. (2013) [50]	Graph-based RDF-3x	Graph partitioning+ Query workload aware distribution	—	Cost-aware query optimization, sub-query sorting according to join key	RDF-3x Query Decomposition, SPARQL subqueries to RDF-3x, Pipelining & parallel processing on coordinator
Galarrága et al. (2014) [34]	Triple Table RDF-3x	Query-workload aware Partitioning	—	Heuristic based distributed plan generation, RDF-3x cost model	RDF-3x SPARQL subqueries to RDF-3x
Tripathi et al. (2014) [104]	Triple Table HDFS	Hash-based partitioning (on subject)	—	Triple pattern reordering	MapReduce Clause-Iteration approach
Wang et al. (2015) [105]	Triple Table RDF-3x	Query-workload aware partitioning	—	Selectivity based triple pattern reordering, Query Rewriting	RDF-3x+MapReduce Query Division & mapreduce job-based processing
Naacke et al. (2016)(2017) [77][76]	Triple Table Spark (standalone)	Hash-based partitioning (on subject)	—	Cost-based join optimization	Spark API SPARQL RDD, SQL, DF, Hybrid RDD, Hybrid DF
Schätzle et al. (2016) [99]	Binary Table HDFS (Parquet)	Extended Vertical (ExtVP) partitioning	—	Filter pushing, Selectivity factor, Semi-join reductions, Results materialization, Selectivity-based triple pattern reordering	SparkSQL Triple pattern mapping
Balaji et al. (2016) [14]	Graph-based Spark (standalone)	Hash-based partitioning (on vertex)	—	Prune set	Spark API Distributed graph path query processing
Madkour et al. (2018) [71]	Binary Table HDFS (Parquet)	Semantic Vertical partitioning (SemVP) +Query-workload aware distribution	—	Semantic filter, Bloom filter, Budgeting mechanism with cost model, Statistics-based triple pattern reordering	Spark SQL Semantic filter read for query answering
Cossu et al. (2018) [24]	Mixed HDFS (Parquet)	Horizontal partitioning (on subject)	—	Statistics-based join reordering	Spark SQL Join Tree format
Mammo et al. (2015) [72]	Triple Table HDFS	compares vertical & horizontal partitionings	—	—	Presto SPARQL queries converted to SQL are run against HDFS
Curé et al. (2015) [26]	Triple Table HDFS	Hash-based Partitioning+ Query-workload aware distribution	—	—	Spark API Query translated to Spark program
Lee et al. (2013) [62]	Triple Table RDF-3x	Semantic hash partitioning	—	—	RDF-3x+MapReduce Partition-aware distributed query processing i.e. intra & inter-partition
Rohloff et al. (2010) [93]	Triple Table HDFS	Hash-based partitioning (on subject)	—	—	MapReduce Clause-Iteration approach
Leng et al. (2017) [64]	Triple Table RDF-3x	Graph Partitioning	—	—	RDF-3x+MapReduce Parallel query processing
Atashkar et al. (2017) [10]	Triple Table MySQL+Spark (standalone)	Query-workload aware partitioning	—	—	Spark API Executes SQL join query on RDD

Table 8

A comparison of Big RDF Frameworks using storage, partitioning, & query processing techniques

Author	Storage Model and Tool	Partitioning	Indexing	Query Optimization	Query Processing Tool and Technique
Sun et al. (2010) [102]	Triple Table HBase	—	6 index scheme; (S_PO, P_SO, O_SP, PS_O, SO_P, PO_S)	Greedy approach	MapReduce MapReduce job-based processing
Stein et al. (2010) [100]	Triple Table SimpleDB	—	1 index scheme; (SPO)	Triple pattern reordering	SimpleDB+Jena Graph API Query division & seperate evaluation
Franke et al. (2011) [33]	Triple Table Hbase+MySQL	—	2 index scheme; (SP_O, OP_S)	Triple pattern reordering, Query Flattening	Hbase+MySQL TriplePattern & BGP matching, SPARQL to SQL translation
Chebotko et al. (2013) [18]	Triple Table Hbase	—	6 index scheme; (S, P, O, SS, OO, SO)	Triple pattern reordering	HBase BGP matching
Papailiou et al. (2012) [86]	Triple Table HBase	—	3 index scheme; (SPO, POS, OSP)	Greedy approach, Hybrid join execution model	HBase+MapReduce Centralized/Distributed execution
Schätzle et al. (2012) [96]	Triple Table HBase	—	2 index scheme; (S_PO, O_PS)	Multiway join optimization	Hbase+MapReduce Map-Side Index Nested Loop Join (MAPSIN join)-based query execution
Papailiou et al. (2013)(2014) [87][85]	Triple Table HBase	—	8 index scheme; (SPO, PSO, POS, OPS, OSP, SOP+2 aggregated indexes)	Hybrid join execution model, join cost model	HBase+MapReduce Centralized/Distributed execution
Choi et al. (2013) [22]	Triple Table HBase	—	3 index scheme; (SPO, OPS, T_com)	Multi-way join processing, Cost-based map-side join	HBase+MapReduce Triple pattern groups (TPG) transfomed into a map job
Xu et al. (2015) [110]	Triple Table HBase	—	3 index scheme; (SPO, POS, OSP)	Hueristics-based join ordering	Spark API Pipeline iterative join processing
Punnoose et al. (2015) [91]	Triple Table Accumulo	—	3 index scheme; (SPO, POS, OSP)	Join reordering, cost-based optimizations, query inferencing, selectivity-based triple pattern reordering	Accumulo API Query expansion & indexed nested loops join operations
Hu et al. (2016) [51]	Triple Table Hbase+Redis	—	6 index scheme; (SP_O, PO_S, SO_P, P_SO, O_SP, S_PO)	Cardinality-based query optimization & triple pattern reordering	Sesame Parallel query execution
Aranda-Andújar et al. (2012) [9]	Triple Table SimpleDB	—	3 index scheme; (SPO, POS, OSP)	—	RDF-3x Index lookups
Li et al. (2015) [65]	Binary Table HBase	—	3 index scheme; (PSO, POS, Path index)	—	Hbase+MapReduce Cascading mapreduce job on Hbase
Ladwig et al. (2011) [61]	Triple Table Cassandra	—	4 index scheme; (SPO, POS, OSP, CSPO)	—	Sesame Index lookups

Table 9

A comparison of Big RDF Frameworks using storage, indexing & query processing techniques

Author	Storage Model and Tool	Partitioning	Indexing	Query Optimization	Query Processing Tool and Technique
Du et al. (2012) [29]	Binary Table Sesame	Hash-based partitioning (on predicate)	3 index scheme; (spoc,posc,cosp)	Cross Product Join	Sesame+MapReduce sub-queries executed in sesame & intermediate results merged by mapreduce jobs
Li et al. (2013) [66]	Triple Table HDFS	Horizontal partitioning	3 index scheme; (S, PO, OS)	greedy matching strategy, pushdown high-selectivity joins	MapReduce Pair-wise joins on filtered triples, Partition-centric identification join strategy
Mutharaju et al. (2013) [75]	Graph-based MongoDB	Graph Partitioning +Hash-based Partitioning (on subject)	2 index scheme (SP_O,PO_S)	Statistics-based join reordering, Triple pattern reordering	MongoDB Parallel processing and Pipelining
Zeng et al. (2013) [112]	Graph-based Trinity	Random Hash Partitioning	4 index scheme; (SPO, OPS, PS, PO)	Cost model, Exploration order optimization using dynamic programming	Trinity Query Decomposition, Message Passing & Graph Exploration
Gu et al. (2014) [39]	Triple Table HBase+Redis	Hash partitioning (on key)	3 index scheme; (SPO, POS, OSP)	Selectivity & statistics based triple pattern reordering	Sesame Parallel processing & single triple pattern matching
Gurajada et al. (2014) [41]	Hybrid —	Graph Partitioning (using METIS)+ Horizontal hash Partitioning	6 index scheme; (SPO, SOP, PSO, OSP, OPS, POS)	Triple pattern reordering using dynamic programming, Exploratory Plan optimization	Master-slave Asynchronous message passing & join-ahead pruning
Chen et al. (2016) [19]	Graph-based HDFS	Hash-based partitioning (on subject)	5 index scheme; (C,R,CR, RC,CRC)	Cost estimation, Type restrictive SPARQL (TR-SPARQL), Selectivity-based triple pattern reordering	Spark API RDSG-based iterative query model
Oh et al. (2015) [81]	Triple Table HBase	Graph partitioning (using METIS)	1 index scheme; (tuni having both S_PO & O_PS)	Heuristics-based join reordering, optimizing query plan using partition information	Hbase+MapReduce Matching RDF data against TPGâĂŹs
Harbi et al. (2015) [43]	Triple Table —-	Hash partitioning (on subject & predicate)+ Workload-aware partitioning	3 index scheme; (P, PS, PO)	locality-aware query optimizer, data statistics used for global query planning	Master-Slaves Message passing, Distributed /Parallel query execution modes
Wylot et al. (2016) [108]	Hybrid —	Workload-based adaptive partitioning	3 index scheme; (Key, type and molecules index)	Template lists	Master-workers Parallel query execution/ Adaptive query execution (Join at master or distributed hash-join)
Rajith et al. (2016) [92]	Triple Table PostgreSQL	Hash partitioning (on subject & object)	2 index scheme (Dual Clustered indexing; 4 clustered indexes for each subject and object table)	Frequency histograms and cardinality estimates	PostgreSQL (query converted to SQL) Rule-based query execution, Parallel sub-query execution
Guo et al. (2015) [40]	Triple Table HDFS	Data centric partitioning	1 index scheme; CCP (Classes Correlations with Property)	Heuristics-based triple pattern reordering, caching, advance filtering of irrelevant triples	Spark+MemSQL API Pattern matching
Khadilkar et al. (2012) [60]	Triple Table HBase	Compares VP, hash etc. partitioning	Indexing strategies like 2, 3 and 6 index schemes are used	—	Jena Querying the storage layout for triples that matches a given triple pattern

Table 10
A comparison of Big RDF Frameworks using storage, partitioning, indexing & query processing techniques

6. Research Challenges

On the basis of the exhaustive literature survey above, a few major research gaps that we have identified have been discussed below:

(i) **Selecting a suitable storage model for Big RDF data:** All the different storage models for RDF discussed above i.e. Triple-based and Graph-based have some limitations. So, selecting a storage structure that is scalable and at the same time suitable for all query types is a tough task. Chuttur et al. [23] discussed the many schemes for RDF storage and identified that selecting a suitable scheme is dependent on many factors. Patchigolla [88] emphasized upon the necessity of selecting a suitable RDF store based on factors such as performance, scalability etc. and depending upon the nature of RDF data that an application requires.

(ii) **Selecting an appropriate partitioning st-rategy:** There is no single partitioning strategy that suits all types of RDF data and all query types. This challenge has been discussed by Pan et al. [84], Akhter et al. [6], Wang et al. [105]. Abdelaziz et al. [3] suggested that a promising research direction is to match the partitioning strategies to RDF graphs instead of using a single partitioning strategy that suits all RDF data graphs and query types.

(iii) **Partitioning cost:** Most of the Big RDF frameworks that use sophisticated partitioning heuristics for Big RDF partitioning suffer from the high costs of preprocessing. While some other which use simple partitioning heu-ristics such as hash partitioning incur high communication cost during query evaluation. Thus, it can be seen that most of these frameworks pay a high cost of data partitioning without taking into regard the query workload. Some of the authors who identified the issues and overhead associated with partitioning large scale RDF data are Harbi et al. [43][44][7], Abdelaziz et al. [3].

(iv) **High communication and I/O cost in distributed query processing:** In Big RDF systems, the large scale RDF datasets are processed by partitioning the data among many nodes or machines. The SPARQL queries are also divided into multiple subqueries and are executed seperately. The distributed processing of these subqueries requires cross node or inter-partition coordination and high amount of data transfer. During query evaluation this I/O and network communication cost is a heavy burden on the system. Abdelaziz et al. [3], Lee et al. [62], Chen et al. [19], Harbi et al. [43][44][7], Peng et al. [89], Chawla et al. [17] identified that for improving SPARQL query performance over Big RDF data it is important to minimize the inter-node or network traffic thereby minimizing communication costs.

(v) **Fault Tolerance:** One main problem with these Big RDF frameworks is the failure of compute nodes. These frameworks rely on the fault tolerance of the underlying cloud computing tools such as Hadoop. For the Big RDF frameworks using a master-slave model the issue is failure of a single compute node or it might become disconnected from the network. Goasdoué et al. [35] discussed that some of the triples from Big RDF data in case of node failure in a distributed scenario. Thus, fault tolerance is a big challenge in such scenario. Some of the authors like Naacke et al. [77], Hassan et al. [48], Gu et al. [39] identified that Big RDF data management is expected to meet some properties such as scalability, fault tolerance etc.

(vi) **Scalable SPARQL query optimization and processing:** Most Big RDF systems use the traditional relational database solutions for SPARQL query optimization without taking into consideration the structure and size of Big RDF data. Secondly, some of these systems such as S2RDF[99] collect data statistics during RDF data preprocessing for SPARQL query optimization. In such cases, tables need to be maintained for storing the results of pre-computations thus, consuming extra storage space. Some of the authors who discussed or addressed the issue of scalable SPARQL query optimization and processing are Nguyen et al. [79], Zou et al. [115], Kaoudi et al. [57], Atre et al. [11] and Elzein et al. [30].

(vii) **Index storage, loading, updation and access cost:** Many of the Big RDF systems using an indexing strategy for RDF data storage store the indexed data in a NoSQL database such as HBase on HDFS. However, as HDFS has a default replication of three so an index table is stored three times thus increasing the storage space. Also, loading Big RDF data into HBase is very costly and it consumes many resources. Savnik et al. [94] and Pan et al. [84] identified that if the dataset is updated frequently then the system will rebuilt the index

continually and hence the computational cost of indexing the dataset may exceed the speedup benefit of accessing operations. Leng et al. [63] discussed that some of the issues in storage and indexing of Big RDF data are joins, storage cost and intermediate results. Curé et al. [25] identified that the index proliferation also came at the cost of high memory footprint.

(viii) **Large number of intermediate results:** A lot of intermediate results are produced when the queries are divided into subqueries and are evaluated locally on different nodes. Lee et al. [62] discussed the high inter-node communication costs associated with transferring large intermediate results of queries across multiple nodes. Wang et al. [105] discussed the heavy burden these large intermediate results bring on both the network and I/O communications. This I/O and network communication cost is a heavy burden on the query evaluation. Husain et al. [53] try to address this issue by using a greedy strategy to pick up a join that produces the smallest size of intermediate results.

7. Conclusions and future work

RDF is a data model that provides means for modeling the web resources in a semi-structured manner. This model is gaining widespread momentum for modeling data from various domains such as digital libraries, Semantic Web, social networks etc. In this paper, we observe that most Big RDF systems use the existing solutions rather than developing new ones. Thus, saving the burden of building a scalable Big RDF framework from scratch. We dive deep into each Big RDF framework to get a wider look into all its components i.e. storage, partitioning, indexing, query optimization and processing. A generic Big RDF framework that is an amalgamation of these five components targeted towards a common objective of improving query performance has been discussed. The most commonly used technique in each component of a Big RDF framework and the reason for its frequent usage has been highlighted at the end of each section. The metrics used for performance evaluation are identified and all discussed Big frameworks are compared on the basis of these metrics. The limitations of each framework are discussed and specific research challenges are suggested as future directions. The best technique for storage, partitioning, indexing, query optimization and processing for Big RDF frameworks are highlighted to guide scalable management of Big RDF data. The subject of our future work is to propose a framework in similar direction by overcoming some of the research challenges discussed above.

References

[1] 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*, 18(2):385–406, 2009.

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

[3] 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.

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

[5] G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis, and D. Plexousakis. RDF query answering using apache Spark: Review and assessment. In *34th International Conference on Data Engineering Workshops (ICDEW)*, pages 54–59, Paris, France, 2018.

[6] A. Akhter, A.-C. N. Ngonga, and M. Saleem. An Empirical Evaluation of RDF Graph Partitioning Techniques. In *European Knowledge Acquisition Workshop*, pages 3–18, Nancy, France, 2018.

[7] R. Al-Harbi, Y. Ebrahim, and P. Kalnis. PhD-Store: An adaptive SPARQL engine with dynamic partitioning for distributed RDF repositories. *arXiv preprint arXiv:1405.4979*, 2014.

[8] M. Ali, K. S. Bharat, and C. Ranichandra. Processing RDF using hadoop. In *Advances in Computing and Information Technology*, pages 385–394. Chennai, India, 2013.

[9] A. Aranda-Andújar, F. Bugiotti, J. Camacho-Rodríguez, D. Colazzo, F. Goasdoué, Z. Kaoudi, and I. Manolescu. AMADA: web data repositories in the amazon cloud. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2749–2751, Maui, USA, 2012.

[10] A. H. Atashkar, N. Ghadiri, and M. Joodaki. Linked data partitioning for RDF processing on Apache Spark. In *3th International Conference on Web Research (ICWR)*, pages 73–77, Tehran, Iran, 2017.

[11] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix Bit loaded: a scalable lightweight join query processor for RDF data. In *Proceedings of the 19th international conference on World wide web*, pages 41–50, Raleigh, USA, 2010.

[12] M. Atre and J. A. Hendler. BitMat: A Main Memory Bit-matrix of RDF Triples. In *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, pages 33–49, Washington, USA, 2009.

[13] R. A. Bahrami, J. Gulati, and M. Abulaish. Efficient processing of SPARQL queries over graphframes. In *Proceedings of the International Conference on Web Intelligence*, pages 678–685, Leipzig, Germany, 2017. ACM.

[14] J. Balaji and R. Sunderraman. Distributed graph path queries using spark. In *IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, pages 326–331, Georgia, USA, 2016.

[15] 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 ACM SIGMOD International Conference on Management of Data*, pages 121–132, New York, USA, 2013.

[16] 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, Sardinia, Italia, 2002.

[17] T. Chawla, G. Singh, and E. S. Pilli. HyPSo: Hybrid Partitioning for Big RDF Storage and Query Processing. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, pages 188–194, Kolkata, India, 2019.

[18] A. Chebotko, J. Abraham, P. Brazier, A. Piazza, A. Kashlev, and S. Lu. Storing, indexing and querying large provenance data sets as RDF graphs in apache HBase. In *2013 IEEE Ninth World Congress on Services*, pages 1–8, California, USA, 2013.

[19] X. Chen, H. Chen, N. Zhang, and S. Zhang. SparkRDF: elastic discreted RDF graph processing engine with distributed memory. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, pages 292–300, Singapore, 2015.

[20] J. Cheng, W. Wang, and R. Gao. Massive RDF data complicated query optimization based on MapReduce. *Physics Procedia*, 25:1414–1419, 2012.

[21] L. Cheng and S. Kotoulas. Scale-out processing of large RDF datasets. *IEEE Transactions on Big Data*, 1(4):138–150, 2015.

[22] P. Choi, J. Jung, and K.-H. Lee. RDFChain: Chain Centric Storage for Scalable Join Processing of RDF Graphs using MapReduce and HBase. In *International Semantic Web Conference (Posters & Demos)*, pages 249–252, Sydney, Australia, 2013.

[23] M. Y. Chuttur. Storage Schemes and Query Optimization Techniques for RDF Data. *International Journal of Advanced Research in Computer Science*, 2(1), 2011.

[24] M. Cossu, M. Färber, and G. Lausen. PRoST: distributed execution of SPARQL queries using mixed partitioning strategies. *arXiv preprint arXiv:1802.05898*, 2018.

[25] O. Curé, G. Blin, D. Revuz, and D. Faye. WaterFowl, a Compact, Self-indexed RDF Store with Inference-enabled Dictionaries. *arXiv preprint arXiv:1401.5051*, pages 1–12, 2014.

[26] O. Curé, H. Naacke, M.-A. Baazizi, and B. Amann. HAQWA: a Hash-based and Query Workload Aware Distributed RDF Store. In *The 14th International Semantic Web Conference, ISWC 2015*, pages 1–4, Bethlehem, USA, 2015.

[27] O. Curé, H. Naacke, M.-A. Baazizi, and B. Amann. On the Evaluation of RDF Distribution Algorithms Implemented over Apache Spark. In *11th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, pages 16–31, Bethlehem, USA, 2015.

[28] F. Du, H. Bian, Y. Chen, and X. Du. Efficient SPARQL query evaluation in a database cluster. In *IEEE International Congress on Big Data*, pages 165–172, Santa Clara, USA, 2013.

[29] J.-H. Du, H.-F. Wang, Y. Ni, and Y. Yu. HadoopRDF: A scalable semantic data analytical engine. In *International Conference on Intelligent Computing*, pages 633–641, Huangshan, China, 2012.

[30] N. M. Elzein, M. A. Majid, I. A. T. Hashem, I. Yaqoob, F. A. Alaba, and M. Imran. Managing big RDF data in clouds: Challenges, opportunities, and solutions. *Sustainable Cities and Society*, 39:375–386, 2018.

[31] O. Erling and I. Mikhailov. Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management*, pages 501–519. Springer, 2010.

[32] D. C. Faye, O. Curé, and G. Blin. A survey of RDF storage approaches. *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, 15:11–35, 2012.

[33] C. Franke, S. Morin, A. Chebotko, J. Abraham, and P. Brazier. Distributed semantic web data management in HBase and MySQL cluster. In *IEEE 4th International Conference on Cloud Computing*, pages 105–112, 2011.

[34] L. Galárraga, K. Hose, and R. Schenkel. Partout: a distributed engine for efficient RDF processing. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 267–268, Seoul, Korea, 2014.

[35] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis. CliqueSquare: efficient Hadoop-based RDF query processing. In *BDA'13-Journées de Bases de Données Avancées*, pages 1–28, Nantes, France, 2013.

[36] F. Goasdoué, Z. Kaoudi, I. Manolescu, J.-A. Quiané-Ruiz, and S. Zampetakis. Cliquesquare: Flat plans for massively parallel RDF queries. In *IEEE 31st International Conference on Data Engineering*, pages 771–782, Seoul, Korea, 2015.

[37] G. Gombos, G. Rácz, and A. Kiss. Spar(k)ql: SPARQL evaluation method on Spark GraphX. In *IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 188–193, Vienna, Austria, 2016.

[38] D. Graux, L. Jachiet, P. Geneves, and N. Layaïda. SPARQLGX: Efficient distributed evaluation of sparql with apache spark. In *The 15th International Semantic Web Conference (ISWC)*, pages 80–87, Kobe, Japan, 2016.

[39] R. Gu, W. Hu, and Y. Huang. Rainbow: A distributed and hierarchical rdf triple store with dynamic scalability. In *IEEE International Conference on Big Data (Big Data)*, pages 561–566, Washington DC, USA, 2014. IEEE.

[40] M. Guo and J. Wang. A Distributed Query Method for RDF Data on Spark. In *National Conference on Big Data Technology and Applications*, pages 102–115, Harbin, China, 2015.

[41] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300, Utah, USA, 2014.

[42] 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*, 8(6):654–665, 2015.

[43] R. Harbi, I. Abdelaziz, P. Kalnis, and N. Mamoulis. Evaluating SPARQL queries on massive RDF datasets. *Proceedings of the VLDB Endowment*, 8(12):1848–1851, 2015.

[44] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 25(3):355–380, 2016.

[45] S. Harris and N. Gibbins. 3store: Efficient bulk RDF storage. In *1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, pages 1–15, 2003.

[46] S. Harris, N. Lamb, N. Shadbolt, et al. 4store: The design and implementation of a clustered RDF store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, pages 94–109, 2009.

[47] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *Proceedings of the 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC+ ASWC*, pages 211–224, Busan, Korea, 2007.

[48] M. Hassan and S. K. Bansal. RDF Data Storage Techniques for Efficient SPARQL Query Processing Using Distributed Computation Engines. In *International Conference on Information Reuse and Integration for Data Science (IRI)*, pages 323–330, Salt Lake City, USA, 2018.

[49] M. Hassan and S. K. Bansal. Semantic Data Querying over NoSQL Databases with Apache Spark. In *International Conference on Information Reuse and Integration for Data Science (IRI)*, pages 364–371, Salt Lake City, USA, 2018.

[50] K. Hose and R. Schenkel. WARP: Workload-aware replication and partitioning for RDF. In *IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*, pages 1–6, Brisbane, Australia, 2013.

[51] C. Hu, X. Wang, R. Yang, and T. Wo. ScalaRDF: a distributed, elastic and scalable in-memory RDF triple store. In *IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 593–601, Wuhan, China, 2016.

[52] 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.

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

[54] M. F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Data intensive query processing for large RDF graphs using cloud computing tools. In *IEEE 3rd International Conference on Cloud Computing*, pages 1–10, Florida, USA, 2010.

[55] M. F. Husain, J. McGlothlin, L. Khan, and B. Thuraisingham. Scalable complex query processing over large semantic web data using cloud. In *IEEE 4th International Conference on Cloud Computing*, pages 187–194, 2011.

[56] E. G. Kalayci, T. E. Kalayci, and D. Birant. An ant colony optimisation approach for optimising SPARQL queries by reordering triple patterns. *Information Systems*, 50:51–68, 2015.

[57] Z. Kaoudi, K. Kyzirakos, and M. Koubarakis. SPARQL query optimization on top of DHTs. In *International Semantic Web Conference*, pages 418–435, Shanghai, China, 2010.

[58] Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 24(1):67–91, 2015.

[59] B. Kassaie. SPARQL over GraphX. *arXiv preprint arXiv:1701.03091*, pages 1–11, 2017.

[60] V. Khadilkar, M. Kantarcioglu, B. Thuraisingham, and P. Castagna. Jena-HBase: a distributed, scalable and efficient RDF triple store. In *Proceedings of the 2012th International Conference on Posters & Demonstrations Track-Volume 914*, pages 85–88, Boston, USA, 2012.

[61] G. Ladwig and A. Harth. CumulusRDF: linked data management on nested key-value stores. In *The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*, pages 30–42, Bonn, Germany, 2011.

[62] 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.

[63] Y. Leng, Z. Chen, and Y. Hu. STLIS: A Scalable Two-Level Index Scheme for Big Data in IoT. *Mobile Information Systems*, 2016.

[64] Y. Leng, C. Zhikui, F. Zhong, X. Li, Y. Hu, and C. Yang. BRGP: a balanced RDF graph partitioning algorithm for cloud storage. *Concurrency and Computation: Practice and Experience*, 29(14):e3896, 2017.

[65] K. Li, B. Wu, and B. Wang. A Distributed RDF Storage and Query Model Based on HBase. In *International Conference on Web-Age Information Management*, pages 3–15, Shandong, China, 2015.

[66] W. Li, B. Chen, R. Yao, Y. Li, W. Wen, C. Cheung, and W. Li. SHOE: A SPARQL Query Engine Using MapReduce. In *International Conference on Parallel and Distributed Systems*, pages 446–447, Seoul, Korea, 2013.

[67] Y. Luo, F. Picalausa, G. H. Fletcher, J. Hidders, and S. Vansummeren. Storing and indexing massive RDF datasets. In *Semantic search over the web*, pages 31–60. Springer, 2012.

[68] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar: an RDF storage and query system for enterprise resource management. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 484–491, Washington, USA, 2004.

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

[70] A. Macina, J. Montagnat, and O. Corby. A SPARQL distributed query processing engine addressing both vertical and horizontal data partitions. In *32nd Conference on Data Management - Principles, Technologies and Applications (BDA)*, pages 1–11, Poitiers, France, 2016.

[71] A. Madkour, W. G. Aref, and A. M. Aly. SPARTI: Scalable RDF Data Management Using Query-Centric Semantic Partitioning. In *Proceedings of the International Workshop on Semantic Big Data*, pages 1–6, San Francisco, USA, 2018.

[72] M. Mammo and S. K. Bansal. Presto-rdf: Sparql querying over big rdf data. In *Australasian Database Conference*, pages 281–293, Melbourne, Australia, 2015.

[73] B. McBride. Jena: Implementing the rdf model and syntax specification. In *Proceedings of the Second International Conference on Semantic Web-Volume 40*, pages 23–28, Hongkong, China, 2001.

[74] M. Meimaris and G. Papastefanatos. Distance-based triple reordering for SPARQL query optimization. In *IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1559–1562, San Diego, USA, 2017.

[75] R. Mutharaju, S. Sakr, A. Sala, and P. Hitzler. D-SPARQ: distributed, scalable and efficient RDF query engine. In *Proceedings of the 12th International Semantic Web Conference (Posters & Demonstrations Track)-Volume 1035*, pages 261–264, Sydney, Australia, 2013.

[76] H. Naacke, B. Amann, and O. Curé. SPARQL graph pattern processing with apache spark. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, pages 1–7, Chicago, USA, 2017.

[77] H. Naacke, O. Curé, and B. Amann. SPARQL query processing with Apache Spark. *arXiv preprint arXiv:1604.08903*, 2016.

[78] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 19(1):91–113, 2010.

[79] M. D. Nguyen, M. S. Lee, S. Oh, and G. C. Fox. SPARQL Query Optimization for Structural Indexed RDF Data. *Grids.Ucs.Indiana.Edu*, pages 1–21, 2014.

[80] Z. Nie, F. Du, Y. Chen, X. Du, and L. Xu. Efficient SPARQL query processing in mapreduce through data partitioning and indexing. In *Asia-Pacific Web Conference*, pages 628–635, Kunming, China, 2012.

[81] H. Oh, S. Chun, S. Eom, and K.-H. Lee. Job-optimized map-side join processing using mapreduce and hbase with abstract RDF data. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, pages 425–432, Singapore, 2015.

[82] A. Owens, A. Seaborne, N. Gibbins, et al. Clustered TDB: A clustered triple store for Jena. In *18th International World Wide Web Conference (WWW)*, pages 1–10, Madrid, Spain, 2008.

[83] M. T. Özsu. A survey of RDF data management systems. *Frontiers of Computer Science*, 10(3):418–432, 2016.

[84] Z. Pan, T. Zhu, H. Liu, and H. Ning. A survey of RDF management technologies and benchmark datasets. *Journal of Ambient Intelligence and Humanized Computing*, 9(5):1693–1704, 2018.

[85] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF+: High-performance distributed joins over large-scale RDF graphs. In *IEEE International Conference on Big Data*, pages 255–263, California, USA, 2013.

[86] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2RDF: adaptive query processing on RDF data in the cloud. In *Proceedings of the 21st International Conference on World Wide Web*, pages 397–400, Lyon, France, 2012.

[87] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H2RDF+: an efficient data management system for big RDF graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 909–912, Utah, USA, 2014.

[88] V. N. R. Patchigolla. *Comparison of clustered RDF data stores*. PhD thesis, Purdue University, 2011.

[89] P. Peng, L. Zou, M. T. Özsu, L. Chen, and D. Zhao. Processing SPARQL queries over distributed RDF graphs. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 25(2):243–268, 2016.

[90] A. Potter, B. Motik, and I. Horrocks. Querying Distributed RDF Graphs: The Effects of Partitioning. In *10th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, pages 29–44, Riva del Garda, Italy, 2014.

[91] R. Punnoose, A. Crainiceanu, and D. Rapp. SPARQL in the cloud using Rya. *Information Systems*, 48:181–195, 2015.

[92] A. Rajith, S. Nishimura, and H. Yokota. JARS: Join-Aware Distributed RDF Storage. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, pages 264–271, Montreal, Canada, 2016.

[93] 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*, pages 1–5, Nevada, USA, 2010.

[94] I. Savnik and K. Nitta. Design of Distributed Storage Manager for Large-Scale RDF Graphs. In *The Sixth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, pages 1–7, Chamonix, France, 2014.

[95] A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen. S2X: Graph-Parallel Querying of RDF with GraphX. *Biomedical Data Management and Graph Online Querying*, pages 155–168, 2015.

[96] A. Schätzle, M. Przyjaciel-Zablocki, C. Dorner, T. Hornung, and G. Lausen. Cascading Map-Side Joins over HBase for Scalable Join Processing. In *Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+HPCSW)*, pages 59–74, Boston, USA, 2012.

[97] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to pig latin. In *Proceedings of the International Workshop on Semantic Web Information Management*, page 4, Washington, USA, 2011.

[98] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen. Sempala: interactive SPARQL query processing on hadoop. In *International Semantic Web Conference*, pages 164–179, Trentino, Italy, 2014.

[99] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF querying with SPARQL on spark. *Proceedings of the VLDB Endowment*, 9(10):804–815, 2016.

[100] R. Stein and V. Zacharias. RDF on cloud number nine. In *4th Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic*, pages 11–23, Heraklion, Greece, 2010.

[101] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*, pages 595–604, Beijing, China, 2008.

[102] J. Sun and Q. Jin. Scalable rdf store based on hbase and mapreduce. In *3rd international conference on advanced computer theory and engineering (ICACTE)*, pages 633–636, Chengdu, China, 2010.

[103] Y. Tanimura, A. Matono, S. Lynden, and I. Kojima. Extensions to the Pig data processing platform for scalable RDF data processing using Hadoop. In *IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*, pages 251–256, California, USA, 2010.

[104] N. Tripathi and S. Banerjee. SARROD: SPARQL Analyzer and Reordering for Runtime Optimization on Big Data. In *International Conference on Big Data Analytics*, pages 189–196, New Delhi, India, 2014.

[105] 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.

[106] 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.

[107] B. Wu, H. Jin, and P. Yuan. Scalable SAPRQL querying processing on large RDF data in cloud computing environment. In *Joint International Conference on Pervasive Computing and the Networked World*, pages 631–646, Istanbul, Turkey, 2012.

[108] M. Wylot and P. Cudré-Mauroux. Diplocloud: Efficient and scalable management of rdf data in the cloud. *IEEE Transactions on Knowledge and Data Engineering*, 28(3):659–674, 2016.

[109] M. Wylot, M. Hauswirth, P. Cudré-Mauroux, and S. Sakr. RDF data storage and query processing schemes: A survey. *ACM Computing Surveys (CSUR)*, 51(4):1–36, 2018.

[110] Z. Xu, W. Chen, L. Gai, and T. Wang. Sparkrdf: In-memory distributed rdf management framework for large-scale social data. In *International Conference on Web-Age Information Management*, pages 337–349, Shandong, China, 2015.

[111] 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*, 6(7):517–528, 2013.

[112] 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*, volume 6, pages 265–276. VLDB Endowment, 2013.

[113] X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 565–576, Brisbane, Australia, 2013.

[114] X. Zhang, L. Chen, and M. Wang. Towards efficient join processing over large RDF graph using mapreduce. In *International Conference on Scientific and Statistical Database Management*, pages 250–259, Chania, Greece, 2012.

[115] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: answering SPARQL queries via subgraph matching. *Proceedings of the VLDB Endowment*, 4(8):482–493, 2011.