Storage, Indexing, Query Processing, and Benchmarking in Centralized and Distributed RDF Engines: A Survey

Waqas Ali
Department of Computer Science and Engineering, School of Electronic, Information and Electrical Engineering (SEIEE), Shanghai Jiao Tong University, Shanghai, China
waqasali@sjtu.edu.cn

Muhammad Saleem
Agile Knowledge and Semantic Web (AKWS), University of Leipzig, Leipzig, Germany
saleem@informatik.uni-leipzig.de

Bin Yao
Department of Computer Science and Engineering, School of Electronic, Information and Electrical Engineering (SEIEE), Shanghai Jiao Tong University, Shanghai, China
yaobin@cs.sjtu.edu.cn

Aidan Hogan
IMFD; Department of Computer Science (DCC), Universidad de Chile, Santiago, Chile
ahogan@dcc.uchile.cl

Axel-Cyrille Ngonga Ngomo
University of Paderborn, Paderborn, Germany
axel.ngonga@upb.de

ABSTRACT

The recent advancements of the Semantic Web and Linked Data have changed the working of the traditional web. There is significant adoption of the Resource Description Framework (RDF) format for saving of web-based data. This massive adoption has paved the way for the development of various centralized and distributed RDF processing engines. These engines employ various mechanisms to implement critical components of the query processing engines such as data storage, indexing, language support, and query execution. All these components govern how queries are executed and can have a substantial effect on the query runtime. For example, the storage of RDF data in various ways significantly affects the data storage space required and the query runtime performance. The type of indexing approach used in RDF engines is critical for fast data lookup. The type of the underlying querying language (e.g., SPARQL or SQL) used for query execution is a crucial optimization component of the RDF storage solutions. Finally, query execution involving different join orders significantly affects the query response time. This paper provides a comprehensive review of centralized and distributed RDF engines in terms of storage, indexing, language support, and query execution.

PVLDB Reference Format:
. . *PVLDB*, (xxx): xxxx-yyyy, .
DOI:

Keywords: Storage, Indexing, Language, Query Planning, SPARQL Translation, Centralized RDF Engines, Distributed RDF Engines, SPARQL Benchmarks, Survey.

1. INTRODUCTION

Over recent years, the simple, decentralized, and linked architecture of Resource Description Framework (RDF) data has greatly attracted different data providers who store their data in the RDF format. This increase is evident in nearly every domain. For example, currently, there are approximately 150 billion triples available from 9960 datasets[1]. Some huge RDF datasets such as UniProt[2], PubChemRDF[3], Bio2RDF[4] and DBpedia[5] have billions of triples. The massive adoption of the RDF format requires effective solutions for storing and querying this massive amount of data. This motivation has paved the way the development of centralized and distributed RDF engines for storage and query processing.

RDF engines can be divided into two major categories: (1) centralized RDF engines that store the given RDF data as a single node and (2) distributed RDF engines that distribute the given RDF data among multiple cluster nodes. The complex and varying nature of Big RDF datasets has rendered centralized engines inefficient to meet the growing demand of complex SPARQL queries w.r.t. storage, computing capacity and processing [126, 81, 50, 3]. To tackle this issue, various kinds of distributed RDF engines were proposed [40, 33, 49, 85, 50, 103, 104]. These distributed systems run on a set of cluster hardware containing several machines with dedicated memory and storage.

[1]http://lodstats.aksw.org/.
[2]http://www.uniprot.org/.
[3]http://pubchem.ncbi.nlm.nih.gov/rdf/.
[4]http://bio2rdf.org/.
[5]http://dbpedia.org/.

doi:10.20944/preprints202005.0360.v3

Efficient data storage, indexing, language support, and optimized query plan generation are key components of RDF engines:

- **Data Storage.** Data storage is an integral component of every RDF engine. Data storage is dependent on factors like the format of storage, size of the storage and inference supported by the storage format [81]. A recent evaluation [4] shows that the storage of RDF data in different RDF graph partitioning techniques has a vital effect on the query runtime.

- **Indexing.** Various indexes are used in RDF engines for fast data lookup and query execution. The more indexes can generally lead to better query runtime performance. However, maintaining these indexes can be costly in terms of space consumption and keeping them updated to reflect the variations in the underlying RDF datasets. An outdated index can lead to incomplete results.

- **Query Language.** Various RDF engines store data in different formats, thus support various querying languages such as SQL [37], PigLatin [79] etc. Since SPARQL is the standard query language for RDF datasets, many of the RDF engines require SPARQL translation (e.g., SPARQL to SQL) for query execution. Such language support can have a significant impact on query runtimes. This is because the optimization techniques used in these querying language can be different from each other.

- **Query Execution.** For a given input SPARQL query, RDF engines generate the optimized query plan that subsequently guides the query execution. Choosing the best join execution order and the selection of different *join types* (e.g., hash join, bind join, nested loop join, etc.) is vital for fast query execution.

Various studies categorize, compare, and evaluate different RDF engines. For example, the query runtime evaluation of different RDF engines are shown in [81, 3, 96, 21, 6]. Studies like [31, 70] are focused towards the data storage mechanisms in RDF engines. Svoboda et al. [114] classify various indexing approaches used for linked data. The usage of relational data models for RDF data is presented in [91]. A survey of the RDF on the cloud is presented in [58]. A high-level illustration of the different centralized and distributed RDF engines and linked data query techniques are presented in [80]. Finally, empirical performance evaluation and a broader overview of the distributed RDF engines are presented in [3]. According to our analysis, there is no detailed study that provides a comprehensive overview of the techniques used to implement the different components of the centralized and distributed RDF engines.

Motivated by the lack of a comprehensive overview of the components-wise techniques used in existing RDF engines. We present a detailed overview of the techniques used in a total of 77 (the largest to the best of our knowledge) centralized and distributed RDF engines. Specifically, we classify these triples stores into different categories w.r.t storage, indexing, language and query planning. We provide simple running examples to understand the different types. We hope this survey will help readers to get a crisp idea of the different techniques used RDF engines development. Furthermore, we hope that this study will help users to choose the appropriate triple store for the given use-case.

The remaining of the paper is divided into different sections. Section 2 provides vital information on RDF, RDF engines and SPARQL. The section 3 is about related work. Section 4, 5, 6 and 7 reviews the storage, indexing, query language and query execution process. Section 8 explains different graph partitioning techniques. Section 9 explains centralized and distributed RDF engines w.r.t storage, indexing, query language and query execution mechanism. Section 10 discusses different SPARQL benchmarks. Section 11 illustrates research problems and future directions, and section 12 gives the conclusion.

2. BASIC CONCEPTS AND DEFINITIONS

This section contains a brief explanation of RDF and SPARQL. The main purpose of explanation is to establish a basic understanding of the terminologies used in the paper. For complete details, readers are encouraged to look at original W3C sources of RDF[6] and SPARQL[7]. This discussion is adapted from [80, 98, 50, 94].

2.1 RDF

Before going to explain the RDF model, we first define the elements that constitute an RDF dataset:

- **IRI:** The International Resource Identifier (IRI) is a general form of URIs (Uniform Resource Identifiers) that allowing non-ASCII characters. The IRI globally identifies a resource on the web. The IRIs used one dataset can be reused in other datasets to represent the same resource.

- **Literal:** is of string value which is not an IRI.

- **Blank node:** refers to anonymous resources not having a name; thus, such resources are not assigned to a global IRI. The blank nodes are used as local unique identifiers, within a specific RDF dataset.

The RDF is a data model proposed by the W3C for representing information about Web resources. RDF models each "fact" as a set of *triples*, where a *triple* consists of three parts:

- **Subject.** The resource or entity upon which an assertion is made. For subject, IRI (International Resource Identifier) and blank nodes are allowed to be used.

- **Predicate.** A relation used to link resources to another. For this, only URIs can be used

- **Object.** Object can be the attribute value or another resource. Objects can be URIs, blank nodes, and strings.

Thus the RDF *triple* represents some kind of relationship (shown by the predicate) between the subject and object. An RDF dataset is the set of *triples* and if formally defined as follows.

[6]RDF Primer: `http://www.w3.org/TR/rdf-primer/`.
[7]SPARQL Specification: `https://www.w3.org/TR/sparql11-query/`

Table 1: Sample RDF dataset with Prefixes: resource = http://uni.org/resource, schema = http://uni.org/schema rdf = http://www.w3.org/1999/02/22-rdf-syntax-ns#. Colors are used to easily understand the data storage techniques discussed in section 4.

Subject	Predicate	Object
resource:Bob	rdf:type	Person
resource:Bob	schema:interestedIn	resource:SemWeb
resource:Bob	schema:interestedIn	resource:DB
resource:Bob	schema:belongsTo	"USA"
resource:Alice	rdf:type	Person
resource:Alice	schema:interestedIn	resource:SemWeb
resource:SemWeb	rdf:type	Course
resource:SemWeb	schema:fieldOf	"Computer Science"
resource:DB	rdf:type	Course
resource:DB	schema:fieldOf	"Computer Science"

Definition 1 (RDF Triple, RDF Dataset). *I, B, and L (IRIs, Blank nodes, and Literals, respectively) are considered as disjoint infinite sets. Then an instance $< s, p, o > \in (I \cup B) \times I \times (I \cup B \cup L)$ is called* RDF triple, *where s is the* subject, *p is the* predicate *and o is an* object. *An* RDF dataset *D is a set of RDF triples $D = \{< s_1, p_1, o_1 >, \dots, < s_n, p_n, o_n >\}$.*

An example RDF dataset, repenting information about a university student is shown in Table 1.

RDF models data in the form of a directed labelled graph where resources are represented as nodes and the relationship between resources are represented as a link between two nodes. For a triple $< s, p, o >$, a node is created for subject s, another node is created for object o, and a directed link with labeled-predicate p is crated from s to o.

An RDF graph can be defined as below [80].

Definition 2 (RDF Graph). *An RDF graph is a tuple of six things $G = \langle V,\ L_V, f_V,\ E,\ L_E,\ f_E \rangle$, where,*

1. $V = V_c \cup V_e \cup V_l$ is a collection of vertices that correspond to all subjects and objects in RDF data, where V_c, V_e, and V_l are collections of class vertices, entity vertices, and literal vertices, respectively

2. L_V is a collection of vertex labels.

3. A vertex labelling function $f_V : V \to L_V$ is a bijective function that assigns to each vertex a label. The label of a vertex $u \in V_l$ is its literal value, and the label of a vertex $u \in V_c \cup V_e$ is its corresponding URI.

4. $E = \{\overrightarrow{u_1,\ u_2}\}$ is a collection of directed edges that connect the corresponding subjects and objects.

5. L_E is a collection of edge labels.

6. An edge labelling function $f_E : E \to L_E$ is a bijective function that assigns to each edge a label. The label of an edge $e \in E$ is its corresponding property.

Figure 1 shows the corresponding RDF graph of the sample RDF dataset given in Table 1.

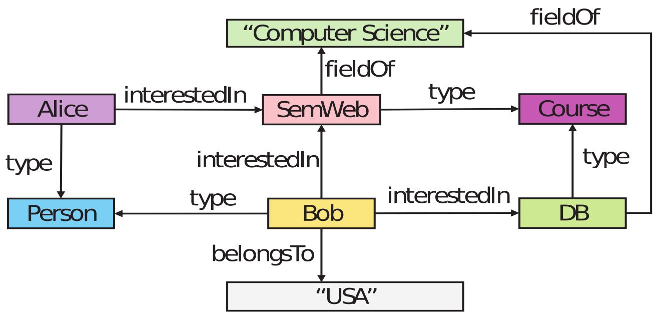


Figure 1. Pictorial representation of graph

2.2 SPARQL

SPARQL [47] is the standard querying language to query RDF data. The basic building units of the SPARQL queries are triple pattern and Basic Graph Pattern (BGP). A triple pattern is like an RDF triple except that each of the subjects, predicate and object may be a variable. A set of triple patterns constitute a BGP[8] and is formally defined as follows.

Definition 3. *(Triple Pattern and Basic Graph Pattern): Assume there are infinite and pairwise disjoint sets I (set of IRIs), B (set of blank nodes), L (set of literals) and V (set of variables). Then, a tuple from $(I \cup V \cup B) \times (I \cup V) \times (I \cup L \cup V \cup B)$ is a triple pattern. A sequence of triple patterns with optional filters is considered a single BGP. As per the specification of BGPs, any other graph pattern (e.g.,* `UNION`, `MINUS`, *etc.) terminates a basic graph pattern.*

Any BGP of a given SPARQL query can be represented as *directed hypergraph* (DH) [98], a general form of a directed graph in which a hyperedge can be used to join any number of vertices. This representation shows that every hyperedge can capture a triple pattern. This representation of hyperedge makes the subject of the triple to becomes the source vertex, and target vertices are shown as the predicate and object of the triple pattern. The hypergraph represented of the SPARQL query is shown in Figure 2. Unlike a common SPARQL representation where the subject and object of the triple pattern are connected by a predicate edge, the hypergraph-based representation contains nodes for all three components (i.e., subject, predicate, object) of the triple patterns. This representation allows to a capture a joins with predicates of triple patterns involved. The SPARQL representation of the hyperedge is defined as follows:

Definition 4 (Directed hypergraph of a BGP). *The basic graph pattern (BGP) B in hyperedge representation is shown as a directed hypergraph $HG = (V, E)$ whose vertices are all the components of all triple patterns in B, i.e., $V = \bigcup_{(s,p,o) \in B} \{s, p, o\}$, and that contains a hyperedge $(S, T) \in E$ for every triple pattern $(s, p, o) \in B$ such that $S = \{s\}$ and $T = (p, o)$.*

SPARQL query represented as DH is represented as a UNION of query BGPs.

Following features of SPARQL queries are defined on the basis of DH representation of SPARQL queries:

[8]BGP https://www.w3.org/TR/sparql11-query/#BasicGraphPatterns

```
SELECT DISTINCT * WHERE
{
 ?student :interestedIn ?course .
 ?student :belongsTo    ?country .
 ?student ?p            ?sType .
 ?course  ?p            ?cType .
 ?course  :fieldOf      ?field .
 ?field   :label        ?fLabel .
}
```

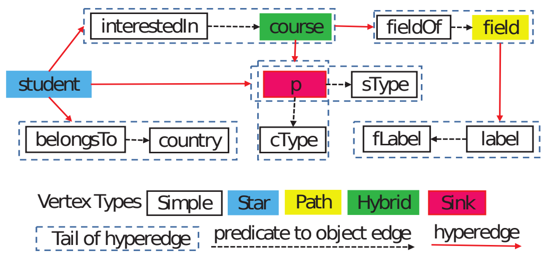


Figure 2: Directed hypergraph representation of a SPARQL query. Prefixes are ignored for simplicity.

Definition 5 (Join Vertex). *For every vertex $v \in V$ in such a hypergraph we write $E_{\text{in}}(v)$ and $E_{\text{out}}(v)$ to denote the set of incoming and outgoing edges, respectively; i.e., $E_{\text{in}}(v) = \{(S,T) \in E \mid v \in T\}$ and $E_{\text{out}}(v) = \{(S,T) \in E \mid v \in S\}$. If $|E_{\text{in}}(v)| + |E_{\text{out}}(v)| > 1$, we call v a* join vertex.

Definition 6 (Join Vertex Types). *A vertex $v \in V$ can be of type* star, path, hybrid, *or* sink *if this vertex participates in at least one join. A* star *vertex has more than one outgoing edge and no incoming edges. A* path *vertex has exactly one incoming and one outgoing edge. A* hybrid *vertex has either more than one incoming and at least one outgoing edge or more than one outgoing and at least one incoming edge. A* sink *vertex has more than one incoming edge and no outgoing edge. A vertex that does not participate in joins is* simple.

Definition 7 (Join Vertex Degree). *Based on the DH representation of the queries the join vertex degree of a vertex v is $JVD(v) = |E_{in}(v)| + |E_{out}(v)|$, where $E_{in}(v)$ resp. $E_{out}(v)$ is the set of incoming resp. outgoing edges of v.*

3. LITERATURE REVIEW

The focus of this section to present studies that discussed the partitioning and storage, indexing, and query processing techniques used in *RDF engines*.

For example, Sakr et al. [91] presented one of the first surveys on the usage of the relational model for RDF data. This survey is all about the use of the relational models for RDF data storage and query processing. A broader overview of the data storage and query processing techniques in centralized and distributed RDF engines is presented in [80]. A survey of the storage of RDF data in relational and NoSQL database stores is presented in [70]. Pan et al. [81] discussed the storage and query processing techniques in centralized and distributed RDF engines. They also compared various benchmarks datasets. A high-level illustration of both storage and query processing in centralized and distributed RDF engines is presented in Sakr et al. [92]. They also discussed various SPARQL benchmarks.

Svoboda et al. [114] discuss different indexing schemes used in centralized and distributed RDF engines. In particular, three types of indexing are discussed, i.e., local, distributed and global. Faye et al. [31] also discussed the storage and indexing techniques in centralized RDF engines. They divided these techniques into non-native and native storage solutions. The non-native solutions make use of the Database Management Systems (DBMS) or other related systems to store RDF data permanently. On the other hand, the native storage solutions store the data close to the RDF model. Thus, such storage solutions avoid the use of DBMS. Rather, the data can be directly stored in different RDF syntaxes[9] such as N-Triples, RDFa, JSON-LD, TriG. A classification of RDF engines is presented in [126]. This study focuses on storage, indexing, query processing mechanisms among RDF engines.

Kaoudi et al. [58] present a survey of RDF systems designed specifically for a cloud-based environment. The focus of the paper to classify the cloud-based RDF engines according to capabilities and implementation techniques. Elzein et al. [28] presented another survey on the storage and query processing techniques used in the RDF engines on the cloud. Janke et al. [54, 57] presented surveys on RDF graph partitioning, indexing, and query processing techniques used in distributed and cloud-based RDF engines. They also discussed some of the available SPARQL benchmarks for RDF engines. A survey and experimental performance evaluation of distributed RDF engines is presented in [3]. The study reviews 22 distributed RDF engines and presents an experimental comparison of 12 selected RDF engines. Reviewed systems belong to categories like graph-based, MapReduce based, and specialized systems.

Yasin et al. [128] discussed the limitations and discrepancies in distributed RDF engines. In particular, they discussed the SPARQL 1.1 support in these engines. Authors in [5] discussed the different categories of RDF engines, including centralized, memory-based, cloud-based, graph-based and binary stores etc. are discussed with their respective examples. In [100], there is a discussion of different mapping techniques to map the RDF data into the NoSQL databases. The respective paper describes the mapping process in different types of NoSQL databases, i.e., key-value and columnar databases etc. through their various examples.

An overview of the surveys, as mentioned earlier on RDF engines is shown in Table 2. The summary indicates that most of the existing studies are focused on the specific components (storage, indexing, query processing) of the centralized or distributed RDF engines. According to our analysis, there exists no detailed study which provides a combined, comprehensive overview of the techniques used to implement the different components of both centralized and distributed RDF engines. Furthermore, previous studies only considered limited RDF engines. We fill this gap by presenting a detailed overview of the techniques used, pertaining to the storage, indexing, language and query execution, both in centralized and distributed RDF engines. We included a complete list of the existing RDF engines as compared to previous studies. In addition, in Section section 10, we discuss different SPARQL benchmarks designed for the performance evaluation of RDF engines. We show the pro and cons of these benchmarks,

[9]RDF Syntaxes `https://www.w3.org/TR/rdf11-concepts/#rdf-documents`

Table 2: An overview of the existing surveys on RDF engines.

Study	Year	Central Focus
Janke et al. [57]	2020	**Partitioning**, **indexing**, and **query evaluation** in **distributed** RDF engines
Santana et al. [100]	2020	**Mapping** strategies for storage of RDF based data into the NoSQL databases
Alaoui [5]	2019	General categorization of different RDF engines
Janke et al. [54]	2018	**Partitioning**, **indexing**, and **query evaluation** in **distributed** and **cloud** RDF engines
Sakr et. al [92]	2018	**Storage** and **query evaluation** in **centralized** and **distributed** RDF engines
Wylot et. al [126]	2018	**Storage** and **indexing** in **centralized** and **distributed** RDF engines
Pan et al. [81]	2018	**Storage** techniques in **centralized** and **distributed** RDF engines
Yasin et al. [128]	2018	Suitability and **query evaluation** of **distributed** RDF engines
Elzein et al. [28]	2018	**Storage** and **query evaluation** in RDF engines on **cloud**
Abdelaziz et al. [3]	2017	**Performance** evaluation of **distributed** RDF engines
Ma et al. [70]	2016	**Storage** techniques in **centralized** RDF engines
Ozsu [80]	2016	General overview of **centralized** and **distributed** RDF engines
Kaoudi et al. [58]	2015	RDF engines on **cloud**
Faye et al. [31]	2012	**Storage** and **Indexing** techniques in **centralized** RDF engines
Svoboda et al. [114]	2011	**Indexing** in **centralized**, **distributed**, and global RDF engines
Sakr et al. [91]	2010	**Storage** and **query evaluation** in SPARQL transnational RDF engines

which will help the user to choose the best representative benchmark for the given use-case.

4. STORAGE

Data storage is an integral component of any Database Management System (DBMS). Efficient data storage is critical for disk space consumption, security, scalability, maintenance, and performance of the DBMS. This section reviews storage mechanisms commonly used in centralized and distributed RDF engines. We divide these mechanisms into five broader categories (ref. Figure 3) namely *Triple Table*, *Property Table*, *Vertical Partitioning*, *Graph-based* data storage solutions, and miscellaneous category comprising of *Key-Value*, *Hbase Tables*, *In-memory*, *Bit Matrix*, storage as an *index permutations* and systems using another system as their the storage component.

4.1 Triple Table

The Triple Table (TT) is the most general approach to save RDF data in a relational style (ref. discussed in section section 9). This style stores all its RDF data in a one large table, which contains three columns, for Subject, Predicate, and Object of the RDF triple. Table 3 shows the TT representation of the RDF Pictorial representation of the sample RDF dataset shown in Table 1.

Table 3: Triple Table representation of the sample RDF dataset shown in Table 1. Prefixes are ignored for simplicity.

Subject	**Predicate**	**Object**
Bob	type	Person
Bob	interestedIn	SemWeb
Bob	interestedIn	DB
Bob	belongsTo	"USA"
Alice	type	Person
Alice	interestedIn	SemWeb
SemWeb	type	Course
SemWeb	fieldOf	"Computer Science"
DB	type	Course
DB	fieldOf	"Computer Science"

To minimize the storage cost and increase query execution performance, the URIs and Strings used in the TT can be encoded as IDs or hash values, and separate dictionaries can be maintained. For example, using a very simple integer dictionary given in Table 4b, the TT table given in Table 3 can be represented as integer TT shown in Table 4a. The use of a dictionary is particularly useful for RDF datasets having many repeating IRIs or literals. However, in RDF datasets, IRIs are more commonly repeated as compared to literals. As such, encoding each distinct literal and assigning a dictionary id may unnecessarily increase the dictionary size. Consequently, this can leads to performance downgrade due to extra dictionary lookups during the query execution. To tackle this problem, some RDF engines (e.g. Jena 2 [121]) only make use of the dictionary tables to store strings with lengths above a threshold. On the one hand, this design aids in applying filter operations directly on the TT. But this multiple storage of string values results in higher storage consumption.

In general, an RDF dataset is a collection of RDF graphs. Thus, an RDF dataset comprises exactly one default graph and zero or more named graphs[10]. Each named graph is a pair consisting of an IRI or a blank node (the graph name), and an RDF graph. Graph names are unique within an RDF dataset. The SPARQL query language lets the user specify the exact named graph to be considered for query execution, thus skipping all others named graphs data to be considered for query processing. Since every RDF triple either belongs to the default graph or specifically named graph, the TT storage can exploit this information and stored the corresponding named graph as the fourth element for each input triple. This specific representation of TT is also called **Quad**, where the table contains four columns; 3 for storing subject, predicate, and object of a triple and the fourth column stores the corresponding named graph of the given triple. According to RDF specification, named graphs are IRIs. For simplicity, let's assume all the triples gave in table Table 3 belongs to named Graph *G1*, then the Quad representation of the given TT is shown in Table 5. The quad representation has been used in many well-known RDF

[10]RDF named graph: `https://www.w3.org/TR/rdf11-concepts/#section-dataset`

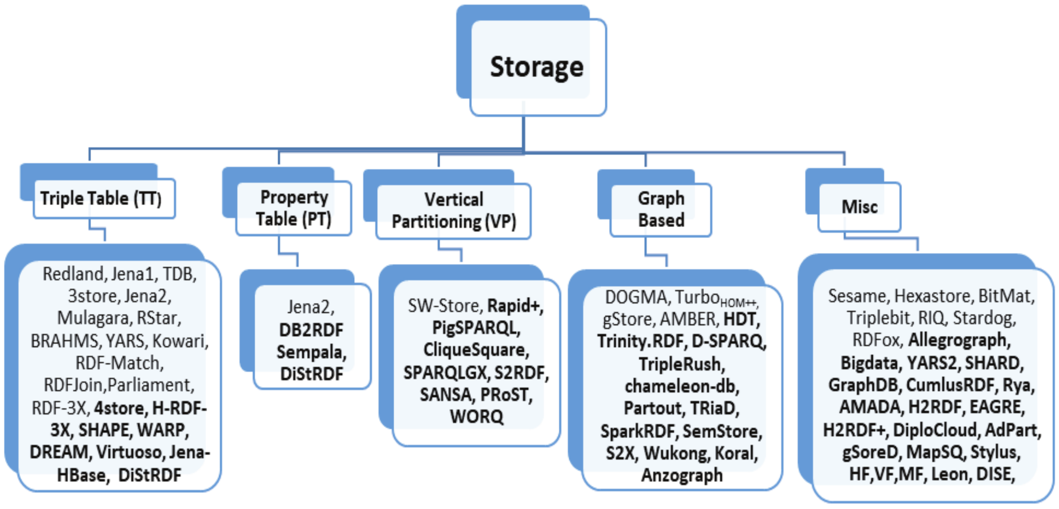

Storage
Triple Table (TT)
Property Table (PT)
Vertical Partitioning (VP)
Graph Based
Misc
Redland, Jena1, TDB, 3store, Jena2, Mulagara, RStar, BRAHMS, YARS, Kowari, RDF-Match, RDFJoin, Parliament, RDF-3X, 4store, H-RDF-3X, SHAPE, WARP, DREAM, Virtuoso, Jena-HBase, DiStRDF
Jena2, DB2RDF Sempala, DiStRDF
SW-Store, Rapid+, PigSPARQL, CliqueSquare, SPARQLGX, S2RDF, SANSA, PRoST, WORQ
DOGMA, TurboHOM++, gStore, AMBER, HDT, Trinity.RDF, D-SPARQ, TripleRush, chameleon-db, Partout, TRiaD, SparkRDF, SemStore, S2X, Wukong, Koral, Anzograph
Sesame, Hexastore, BitMat, Triplebit, RIQ, Stardog, RDFox, Allegrograph, Bigdata, YARS2, SHARD, GraphDB, CumlusRDF, Rya, AMADA, H2RDF, EAGRE, H2RDF+, DiploCloud, AdPart, gSoreD, MapSQ, Stylus, HF,VF,MF, Leon, DISE,

Figure 3: Pictorial representation of Storage in Centralized and Distributed RDF Engines (*RDF Engines in bold text are Distributed engines)

Table 4: Triple Table representation of the TT shown in Table 3 using dictionary encoding.

(a) Integer Triple Table using dictionary

Subject	Predicate	Object
1	5	9
1	6	3
1	6	4
1	7	12
2	5	9
2	6	3
3	5	10
3	8	11
4	5	10
4	8	11

(b) Dictionary

ID	String	ID	String
1	Bob	2	Alice
3	SemWeb	4	DB
5	type	6	interestedIn
7	belongsTo	8	fieldOf
9	Person	10	Course
11	“Computer Science”	12	“USA”

engines[11] such as Virtuoso [30] and 4store [43]. Please note that as per SPARQL specification[12], a SPARQL query may specify the RDF graph to be used for matching by using the `FROM` clause and the `FROM NAMED` clause to describe the RDF dataset. Such queries can be efficiently executed by using Quad tables; as such queries should be executed over the specified named graph and hence skipping triples that belong to other named graphs.

Table 5: Quad representation of the sample RDF dataset shown in Table 1. Prefixes are ignored for simplicity.

Subject	Predicate	Object	Named Graph
Bob	type	Person	G1
Bob	interestedIn	SemWeb	G1
Bob	interestedIn	DB	G1
Bob	belongsTo	"USA”	G1
Alice	type	Person	G1
Alice	interestedIn	SemWeb	G1
SemWeb	type	Course	G1
SemWeb	fieldOf	"Computer Science”	G1
DB	type	Course	G1
DB	fieldOf	"Computer Science”	G1

Summary. The relational DBMS storage style used in TT saves expensive joins between separate tables but incurs expensive self joins. SPARQL queries containing multiple triple patterns applied in this storage style are slow to execute because of the huge number of self joins. Consequently, this might not be a scaleable solution for storing Big Data. Query execution over a single giant table is also sub-optimal. This is because the whole dataset has to be scanned at least once, even if the query is only for a minimal subset of the table. However, this problem can be degraded by using multiple indexes on the TT. For example, in section section 9 we will

[11]Further details given in section section 9.
[12]Specifying RDF datasets: `https://www.w3.org/TR/sparql11-query/#specifyingDataset`

see that TT often comes with various indexes, mostly six triple permutations $\mathbf{P}(< s, p, o >) =$ **SPO, SOP, PSO, POS, OPS, OSP**.

4.2 Property Table

The Property Tables approach aims to lessen the number of joins needed evaluation of for SPARQL Basic Graph Pattern. In this approach, all properties (i.e. predicates) that are expected to be used in combination are stored in one table. A typical property table contains exactly one column to store the subject (i.e. a resource) and n number of columns to store the corresponding properties of the given subject. For example, in our sample RDF dataset shown in Table 1, the subject `DB` has two properties namely `type` and `fieldOf`, thus a typical property table would have three columns: one to store the subject and two to store the corresponding properties.

There are two ways to determine the set of properties grouped together in a property table: (1) make use of the type definitions (rdf:type as shown in Table 1) in the dataset itself, (2) use some clustering algorithm to determine the properties groups. Our sample RDF dataset shown in Table 1 explicitly mentions two `rdf:types` namely *Person* and *Course*. Thus, we can group all properties that belongs to *Person* in one table and all properties that belongs to *Course* in another table. Table 6 shows the corresponding property tables for the RDF dataset shown in Table 1. Table 6 revealed two explicit disadvantages of using property tables:

- **Multi-valued properties**: Multi-valued predicates are common in databases. For example, a person can have more than one contact numbers. In our example, `interestedIn` is multi-valued predicate: *Bob* is interested both in *SemWeb* and *DB* courses. A typical multi-valued predicate will introduce duplicated information in the columns. For example, in Table 6a, the `type` and the country (`belongsTo` predicate) information is duplicated for the subject *Bob*. One way to handle this problem to use the typical database normalization approach, i.e. create separate property tables for multi-valued predicates. Table 7 shows the corresponding properties tables, after creating separate tables for multi-valued predicates.

- **Null values**: It is very common in RDF datasets that certain resources have missing information for some particular predicates. For example, the country of the resource *Alice* is missing in Table 1. Such missing information is typically represented as null values. The low datasets *structuredness* values shown in [99] suggest that many real-world RDF datasets contain missing information for different resources.

Summary. PT performs very well for executing *star joins* (ref. Figure 2) in the query. This is because, a *star* join node is based on a *subject-subject* joins, thus a typical property table will act like a subject-based index for executing such joins. However, it suffers for executing other types of joins e.g, *path*, *hybrid*, and *sink* (ref. ref. Figure 2) used in the SPARQL queries. A *path* join node refers to *subject-object* join, *sink* join node refers to *object-object* join, and *hybrid* join node refers combination of all. The real-world users queries statistics of the four datasets, presented in [93], show that 33% of the real-world queries contain *star* join, 8.79%

Table 6: Property tables representation of the sample RDF dataset shown in Table 1. **Multi-valued predicate `interestedIn` is not treated separately**. The additional row containing majority of the duplicate entries introduced by the multi-valued predicate is highlighted gray. Prefixes are ignored for simplicity.

(a) Property table of the subjects of type `Person`

Subject	**type**	**interestedIn**	**belongsTo**
Bob	Person	SemWeb	“USA"
Bob	Person	DB	“USA"
Alice	Person	SemWeb	null

(b) Property table of the subjects of type `Course`

Subject	**type**	**fieldOf**
SemWeb	Course	“Computer Science"
DB	Course	“Computer Science"

Table 7: Property tables representation of the sample RDF dataset shown in Table 1. **Multi-valued predicate `interestedIn` is treated separately**. Prefixes are ignored for simplicity.

(a) Property table of the subjects of type `Person`, excluding multi-valued predicates

Subject	**type**	**belongsTo**
Bob	Person	“USA"
Alice	Person	null

(b) Property table of the subjects of type `Person` for multi-valued predicate

Subject	**interestedIn**
Bob	SemWeb
Bob	DB
Alice	SemWeb

(c) Property table of the subjects of type `Course`

Subject	**type**	**fieldOf**
SemWeb	Course	“Computer Science"
DB	Course	“Computer Science"

contain *path* join, 6.62% contain *sink* join, 4.51% contain *hybrid* join, and 66.51% contains no join at all. Thus, this approach can produce give efficient results for majority of the queries containing joins between triple patterns. Furthermore, different tools tried to reduce the problems and performance deficiencies associated PT. For example, in [121], PT were used together with a TT, where the PT aims to store the most commonly used predicates. Finally, this approach is sensitive to the underlying schema or data changes in the RDF datasets.

4.3 Vertical Partitioning

Vertical Partitioning (VP) was proposed in [2]. In contrast to PT and TT, a VP stores RDF data in two columns tables form. Subject and object named by the property.The number of tables equals the number of distinct predicates used in the RDF dataset. Since there are four distinct predicates in the sample RDF dataset shown in Table 1, the corresponding vertical tables are shown in Table 8.

Table 8: Vertical partitioning of the sample RDF dataset shown in Table 1. Prefixes are ignored for simplicity.

(a) Predicate `type`

Subject	Object
Bob	Person
Alice	Person
SemWeb	Course
DB	Course

(b) Predicate `interestedIn`

Subject	Object
Bob	SemWeb
Bob	DB
Alice	SemWeb

(c) Predicate `belongsTo`

Subject	Object
Bob	"USA"

(d) Predicate `fieldOf`

Subject	Object
DB	"Computer Science"
SemWeb	"Computer Science"

In contrast to VP, the VT does not suffer from the multi-valued predicates and the missing information that corresponds to null values. The approach is particularly useful for answering SPARQL triple patterns with bound predicates (e.g. ?s p ?o). This is because the predicates tables can be regarding as an index on predicates, hence only a single table needs to considered while answering triple patterns with bound predicates. However, this type of storage is not optimized for answering triple patterns containing unbounded predicates (e.g., s ?p ?o). This is due to the fact since predicate is shown as a variable in the triple pattern, the triple pattern matching will consider all vertical tables. Consequently, may fetched a large portion of intermediate results which will be discarded afterwards.

Summary. VP proves to be efficient in practice for large RDF datasets with many predicates, as it also offers an indexing by predicates.This approach is particularly useful for column-oriented DBMS and is easy to manage in a distributed setups. However, the table sizes can significantly vary in terms of number of rows. As such, some partitions can account for a large portion of the entire graph, leading to workload imbalance. Furthermore, it can cause a lot of I/O resources for answering SPARQL queries with unbound predicates.

4.4 Graph Based Storage

A graph is a natural storage form of RDF data. Various (un)directed, (a) cyclic (multi) graph data structures can be used to store RDF graphs. For example, [133] store RDF graphs as directed signature graphs stored as a disk-based adjacency list table, [15] store as balanced binary tree, and [51] store as multigraphs etc. A very simple labelled, directed signature graph of the Figure 1 is shown in Figure 4. In Graph-based approaches, the given SPARQL query is also represented as graph and sub-graph matching is performed to answer the query.

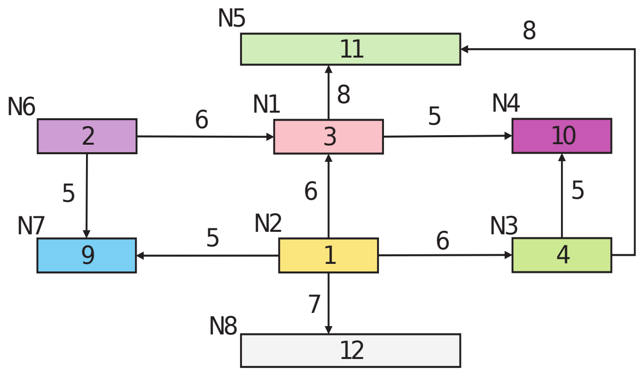


Figure 4. Signature graph representation using dictionary encoding shown in Table 4b

Summary. The main advantage of graph storage of RDF data is that it is the original representation of the RDF data hence and represents the the original semantics of SPARQL. As graph homo-morphism is NP-complete, the sub-graph matching can be costly. In particular, graph storage can raises issues pertaining to the scalability of the RDF engines for large graphs, which can be addressed by indexing of database management techniques.

4.5 Miscellaneous Storage

This category includes multiple storage schemes which are yet not widely used in the state of the art. Main storage schemes are Key-Value based [132], HBase tables based[13] [82], API based like Sesame SAIL Storage And Inference Layer [16], on disk or in memory-based [12], and Bit Matrix-based [129]. We encourage readers to refer to the corresponding papers for the details of these RDF storage schemes.

5. INDEXING

Indexing is one of the important component of the database management systems. The right selection of indexing can significantly improve the query runtime performances. However, indexes need to be updated with underlying changes in the data sets. In particular, if the data changes on regular intervals, too many indexes may slow down the performance. In addition, indexes needs extra disk space for storage. Major types of indexing that we found during the study are given in Figure 5.

- **Indexing By Predicate**

  This type of indexing is found in RDF engines where the data is stored as Vertically Partitioned (VP) tables.

[13]http://hbase.apache.org.

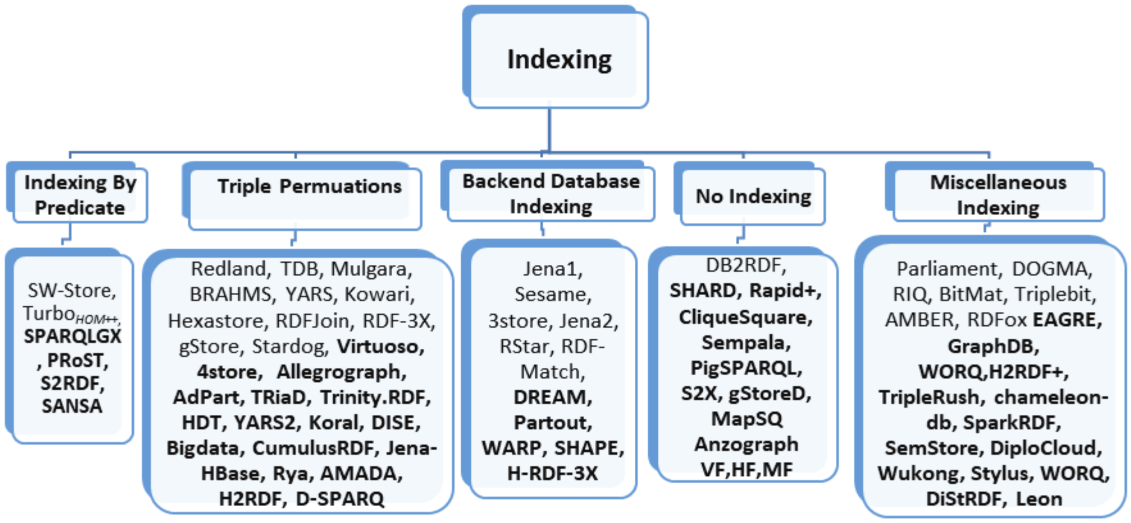

Indexing
Indexing By Predicate
Triple Permuations
Backend Database Indexing
No Indexing
Miscellaneous Indexing
SW-Store, Turbo_HOM++, SPARQLGX, PRoST, S2RDF, SANSA
Redland, TDB, Mulgara, BRAHMS, YARS, Kowari, Hexastore, RDFJoin, RDF-3X, gStore, Stardog, Virtuoso, 4store, Allegrograph, AdPart, TRiaD, Trinity.RDF, HDT, YARS2, Koral, DISE, Bigdata, CumulusRDF, Jena-HBase, Rya, AMADA, H2RDF, D-SPARQ
Jena1, Sesame, 3store, Jena2, RStar, RDF-Match, DREAM, Partout, WARP, SHAPE, H-RDF-3X
DB2RDF, SHARD, Rapid+, CliqueSquare, Sempala, PigSPARQL, S2X, gStoreD, MapSQ, Anzograph, VF,HF,MF
Parliament, DOGMA, RIQ, BitMat, Triplebit, AMBER, RDFox EAGRE, GraphDB, WORQ,H2RDF+, TripleRush, chameleon-db, SparkRDF, SemStore, DiploCloud, Wukong, Stylus, WORQ, DiStRDF, Leon

Figure 5: Pictorial representation of Indexing schemes in Centralized and Distributed RDF Engines (*RDF Engines in bold text are Distributed engines)

Each of the VP table is naturally indexed by predicate. Predicate-based indexing is particularly helpful for answering triple patterns with bound predicates. In such cases, only one VP table is consulted to answer the given triple pattern. However, a SPARQL triple pattern can have 8 different combinations based on bounded or unbounded s, p, o, as shown in Figure 6. As such, only using a single predicate-based index will be less efficient for triple patterns with unbound predicates.

Summary. If data is stored in VP tables, it is automatically indexed by predicate as well. It is a space efficient solution and can be useful for datasets with frequent updates. Furthermore, triple patterns of types $<?s, : p, ?o>, <: s, : p, ?o>, and <?s, : p, : o>$, can directly answered by only consulting a single VP table. However, it is less efficient for tps with predicates as a variable, e.g. $<: s, ?p, ?o>, <?s, ?p, : o>, and <: s, ?p, : o>$.

- **Triple-Permutation Indexing.** As mentioned before, an RDF triple comprises Subject, Predicate, and Object. Figure 7 shows the permutations of an RDF triple ⟨s,p,o⟩. The RDF engines in this category creates indexes pertaining to some or all permutations of the SPO. The goal is to efficiently execute the different types of SPARQL triple patterns shown in Figure 6. Consequently, the Triple-Permutation indexes can be particularly helpful in the efficient execution of triple types of SPARQL joins (ref. Figure 2). The subject, predicate, object permutations can be extended to quad to include the fourth element of named graph or model.

  Figure 8 shows the SPO index for the sample RDF dataset given in Table 1. For each distinct subject s_i in a dataset D, a vector of predicates $V_P = \{p_1^i, \dots, p_n^i\}$ in maintained; and for each element $p_j^i \in V_P$, a separate list $L_O = \{o_1^{i,j}, \dots, o_k^{i,j}\}$ of the corresponding objects maintained. The SPO index can directly answer triple patterns of type $<: s, : p, ?o>$. Similarly, SOP can directly answer triple pattern of type $<: s, ?p, : o>$ and so on. Finally, both SPO and sop can be used to answer triple pattern of type $<: s, ?p, ?o>$. The same approach can be followed for other type of triple-permuted indexes. The use of dictionary is helpful to reduce the storage cost of such indexes.

  Summary. The triple-permutation indexes solve the aforementioned issue of unbound predicates, associated with predicate-based indexing. The triple-permutation indexes can directly answer all types of triple patterns shown in Figure 6, and hence avoiding expensive self joins on the large TT. However, they suffers from a severe storage overhead and are expensive to be maintained for RDF data with frequent changes.

- **Backend Database Indexing** Certain RDF engines, e.g. Jena1, Jena2, and Sesame, etc. make use of the existing DBMS as backend for storing RDF data. Usually, there are multiple indexes available from that backend DBMS which are utilized as an indexes. For example, in Oracle DBMS various index including b-tree, function-based reverse key etc. are available. PostgreSQL[14] provides several indexes e.g., b-tree, hash, GiST, SP-GiST, GIN and BRIN.

  Summary. Using existing DBMS systems avoid the extra work of creating indexes. The available indexes in such DBMS are already mature and optimized for the data these DBMS are designed. However, due to different semantics of RDF data, they available indexes or storage solutions in the existing DBMS may not be optimized for RDF datasets.

- **No Indexing** Some centralized and distributed RDF engines do not create any indexing scheme at all. This could be due to the storage solution they used can be

[14]PostgreSQL index types: `https://www.postgresql.org/docs/9.5/indexes-types.html`

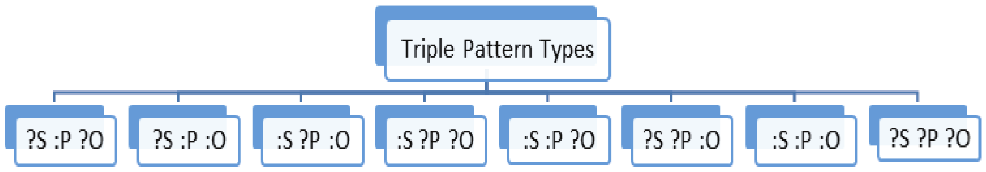


Figure 6. Different types of SPARQL triple patterns based on (un)bound subject S , predicate P and object O. The ":" refers to bound and "?" refers to unbound subject, predicate or object of a triple pattern.

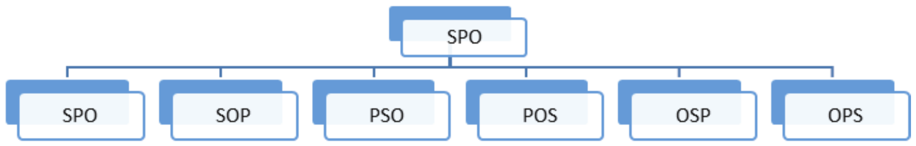


Figure 7. Common Triple-Permutation indexes in RDF-3X and Hexastore.

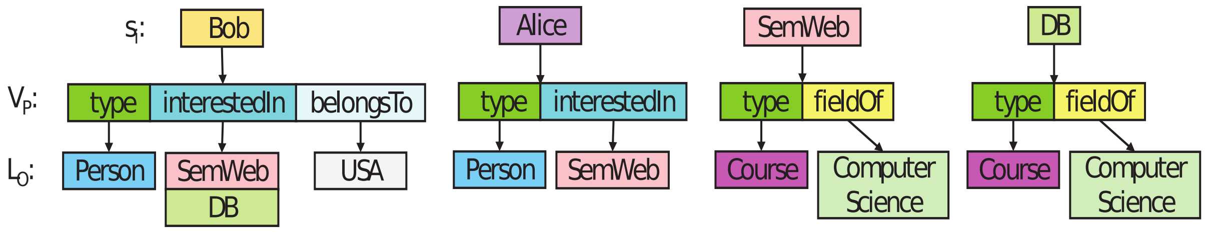


Figure 8: Simple SPO index for the sample RDF dataset given in Table 1.

working as natural index. For example, the graph-based storage solutions serve as natural index as well; where the query execution is reduced to sub-graph matching problem.

- **Miscellaneous Indexing**

  This type of indexing contains multiple types, i.e., Local predicate and global predicate indexing [131], Space-filling [132], array indexing [64] and Hbase tables[15] [82] indexing scheme etc. We encourage readers to refer to the corresponding papers for the details of these indexing schemes.

6. LANGUAGE SUPPORT

Language is an interface through which any data repository can be queried. RDF data can be directly queried via SPARQL. However, a translation of SPARQL to other querying language is required if the data is not stored in RDF format. Summary of query languages in centralized and distributed RDF engines is shown below in Figure 9.

The efficient translation of SPARQL into other querying languages is one of the major optimization step in such engines. Some RDF engines use different API's for querying purposes as well. Different types of the query languages found during the study are given below.

1. **SPARQL**: SPARQL is the standard query language for RDF based dataa. Directly using SPARQL on top of RDF can be much faster, as the underlying semantics of RDF and SPARQL can be utilized towards better runtime performance. In addition, the aforementioned triple-permutation indexes can be leveraged to further optimize the query execution. Majority (ref. Figure 9) of the RDF engines make use of the direct SPARQL execution on top of RDF data. However, storing Big RDF datasets in format that can be directly queried via SPARQL is still a challenging task.

2. **SPARQL Translation**: As mentioned before, certain RDF engines make use of the existing DBMS as backend for storing RDF data. The relational DBMS systems (PostgreSQL, Oracle etc.) are the most popular among them. Since SPARQL is not directly supported by these DBMS, a translation of SPARQL to DMBS-supported-language is required for query execution. Most of the RDF engines in this category translate SPARQL to SQL using existing translation tools e.g., Sparklify [110], Ontop [18] etc.

3. **Other Languages**: Some RDF engines also use their own query language other than SPARQL i.e., RDF Data Query Language (RDQL) [72], Interactive Tucana Query Language (iTQL) etc. Some RDF engines are used as a libraries to be used with different applications and offer full or part of RDF functionality [12].

7. QUERY EXECUTION

Different RDF engines employ different query execution strategies. The overall goal is to devise an optimized query execution plan that leads to fast query execution. This is generally achieved by using different query optimization strategies, e.g., the selection of query planning trees (e.g. bushy tree vs. left-dept tree), the lowest cardinality joins should be executed first, parallel execution of different joins, minimization of intermediate results, use of different joins (hash, bind, merge sort etc.), filter operations are push down in the query plan etc. As such, categorizing the complete surveyed systems according to their query execution strategies is rather hard; a variety of optimization strategies are used in state-of-the-art RDF engines. A broader categories (ref. Figure 10) of the query execution strategies are explained below.

1. **Statistics Based**

   This type of query plans is based on the cardinality (estimated or actual) of triple patterns and join between triple patterns. The cardinality estimations are may be performed by using stored statistics. The actual cardinalities can be obtained at runtime by using different queries, executed on the underlying dataset. The goal is to execute the lowest cardinality join first, followed by the next lowest cardinality join and so on. The advantage of executing the lowest cardinality joins first is that the intermediate results for the next join could be significantly reduced, hence the join operations are performed quickly. The cost of cardinality of a particular join is estimated by using the stored statistics as index.

2. **Translation Based** Those RDF engines that make use of the existing DBMS systems can take advantage of different optimization steps used, already implemented in these DBMS. For such systems, the SPARQL query language is translated into another language supported by the underlying DBMS. Thus, additional optimization can be applied during the translation, e.g., the ordering of triple patterns, the use of filters etc.

3. **API Based**

   Some RDF engines are used as a library (e.g. Jena [72], Sesame [16] to store and query datasets. Query execution is dependent on the Application-specific procedure to generate efficient query plans.

4. **Subgraph Matching Based**

   RDF engines in this category exploit the graph nature both for RDF data and the corresponding SPARQL query. Once both data and query is represented as graph, the query execution problem is reduced to the subgraph matching problem. query execution.

5. **Miscellaneous** In this category, different types of query execution models exist. For example index lookups combined with different joins, heuristic-based query processing and join ordering etc.

8. RDF GRAPH PARTITIONING

In distributed RDF engines, the data is distributed among cluster nodes. This partitioning of big data among multiple data nodes helps in improving systems availability, ease of maintenance, and overall query processing performances. Formally, the problem of RDF graph partitioning is defined as below.

[15]https://hbase.apache.org/

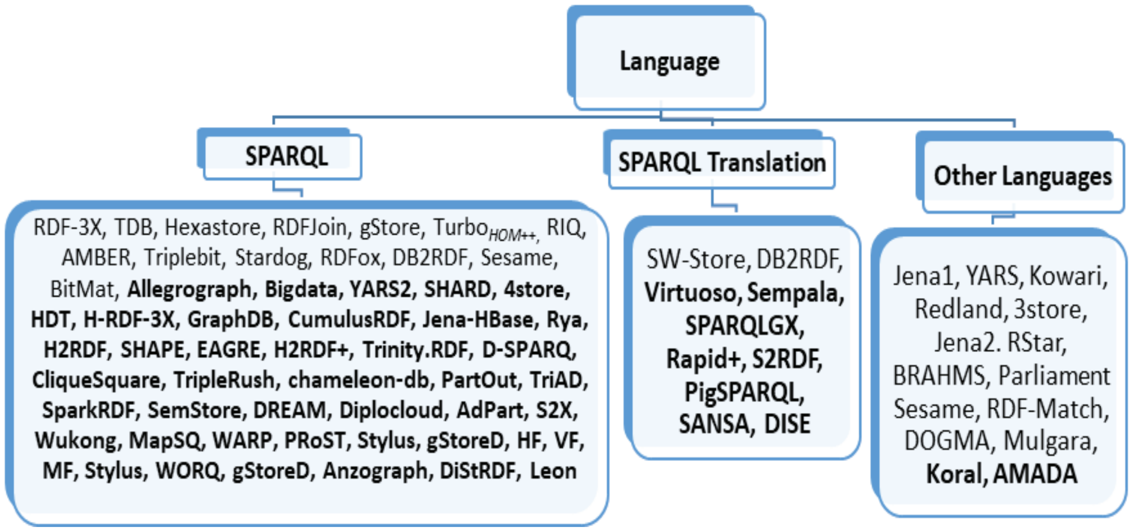

Language
SPARQL
SPARQL Translation
Other Languages
RDF-3X, TDB, Hexastore, RDFJoin, gStore, Turbo HOM++, RIQ, AMBER, Triplebit, Stardog, RDFox, DB2RDF, Sesame, BitMat, Allegrograph, Bigdata, YARS2, SHARD, 4store, HDT, H-RDF-3X, GraphDB, CumulusRDF, Jena-HBase, Rya, H2RDF, SHAPE, EAGRE, H2RDF+, Trinity.RDF, D-SPARQ, CliqueSquare, TripleRush, chameleon-db, PartOut, TriAD, SparkRDF, SemStore, DREAM, Diplocloud, AdPart, S2X, Wukong, MapSQ, WARP, PRoST, Stylus, gStoreD, HF, VF, MF, Stylus, WORQ, gStoreD, Anzograph, DiStRDF, Leon
SW-Store, DB2RDF, Virtuoso, Sempala, SPARQLGX, Rapid+, S2RDF, PigSPARQL, SANSA, DISE
Jena1, YARS, Kowari, Redland, 3store, Jena2. RStar, BRAHMS, Parliament Sesame, RDF-Match, DOGMA, Mulgara, Koral, AMADA

Figure 9: Pictorial representation of Query languages in Centralized and Distributed RDF Engines (*RDF Engines in bold text are Distributed engines)

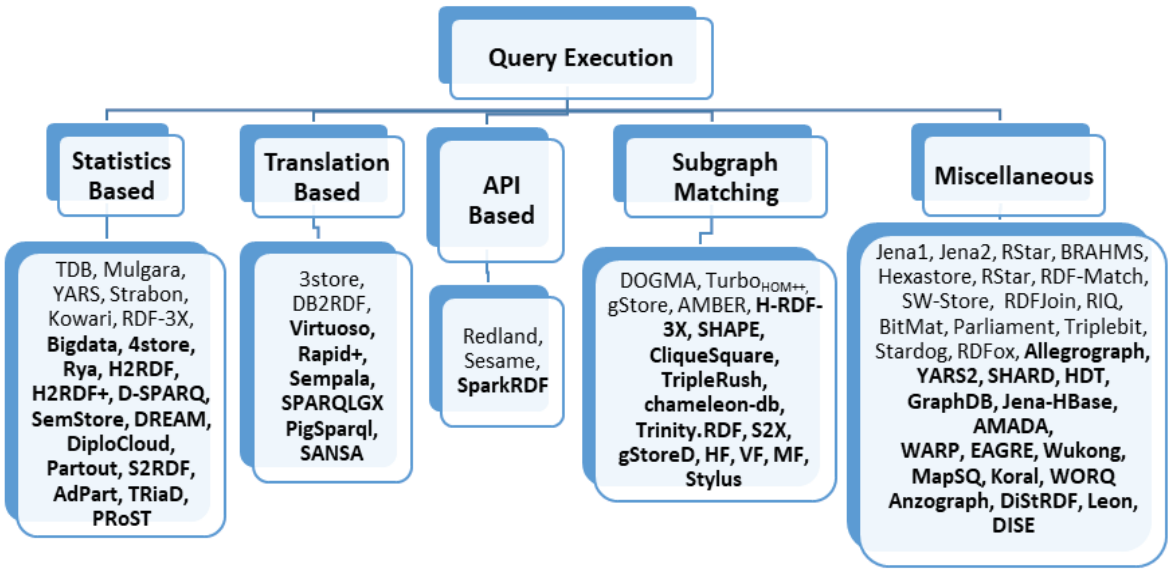

Query Execution
Statistics Based
Translation Based
API Based
Subgraph Matching
Miscellaneous
TDB, Mulgara, YARS, Strabon, Kowari, RDF-3X, Bigdata, 4store, Rya, H2RDF, H2RDF+, D-SPARQ, SemStore, DREAM, DiploCloud, Partout, S2RDF, AdPart, TRiaD, PRoST
3store, DB2RDF, Virtuoso, Rapid+, Sempala, SPARQLGX PigSparql, SANSA
Redland, Sesame, SparkRDF
DOGMA, Turbo HOM++, gStore, AMBER, H-RDF-3X, SHAPE, CliqueSquare, TripleRush, chameleon-db, Trinity.RDF, S2X, gStoreD, HF, VF, MF, Stylus
Jena1, Jena2, RStar, BRAHMS, Hexastore, RStar, RDF-Match, SW-Store, RDFJoin, RIQ, BitMat, Parliament, Triplebit, Stardog, RDFox, Allegrograph, YARS2, SHARD, HDT, GraphDB, Jena-HBase, AMADA, WARP, EAGRE, Wukong, MapSQ, Koral, WORQ Anzograph, DiStRDF, Leon, DISE

Figure 10. Types of Query Execution in Centralized and Distributed RDF Engines(*Bold represents the Distributed RDF Engines)

Definition 8 (RDF Graph Partitioning Problem). *Given an RDF graph with vertices (V) and edges (E),* $G = (V, E)$*, divide the graph G into n sub-graphs* $G_1, \ldots G_n$ *such that* $G = (V, E) = \bigcup_{i=1}^{n} G_i$.

RDF graph partitioning techniques can be divided into two major categories:

- **Horizontal Partitioning.** It is row-wise distribution of data into different partitions. It is also called database sharding. In RDF each row of a TT represents a triple, the triple-wise distribution of complete dataset is regarded as horizontal partitioning.
- **Vertical Partitioning.** It is column-wise distribution of data into different partitions and involves creating tables with fewer columns and using additional tables to store the remaining columns. In the context of RDF dataset, each triple represents a row with three columns namely subject, predicate and object. Hence, distribution by any of these columns is regarded as vertical RDF partitioning. Famous example of vertical partitioning by predicate column is already discussed in 4.

A recent empirical evaluation [4] of the different RDF graph partitioning showed that the type of partitioning used in the RDF engines have a significant impact of the query runtime performance. They conclude that the data that is queried together in SPARQL queries should be kept in same node, thus minimizing the network traffic among data nodes. The Figure 11 shows categories of the partitioning techniques found in distributed RDF engines. Please note that all the engines under workload-based, hash-based and graph-based categories are the examples of horizontal partitioning.

Now we define each of the category given in Figure 11. We explain commonly used [54, 60, 104, 95] graph partitioning techniques by using a sample RDF graph shown in Figure 12[16]. In this example, we want to partition the 11 triples into 3 partitions namely green, red, and blue partitions.

Range Partitioning: It distributes triples based on certain range values of the partitioning key. For example, create a separate partition of all RDF triples with Predicate `age` and object values between 30 and 40. In our motivating example, let the partition key is the triple number with partitions defined according the following ranges: first partition is created for all the triples in the range [1,4], a second partition is created for all the triples in the range [5,8], and third partition is created for all the triples in the range [9,11].

Workload-Based Partitioning: The partitioning techniques in this category make use of the query workload to partition the given RDF dataset. Ideally, the query workload contains real-world queries posted by the users of the RDF dataset which can be collected from the query log of the running system. However, the real user queries might not be available. In this case the query work load can either be estimated from queries in applications accessing the RDF data or synthetically generated with the help of the domain experts of the given RDF dataset that needs to be partitioned.

Hash-Based Partitioning: There are three techniques used in this category:

- **Subject-hashed.** In this technique hash function is applied on the subject of the triple and based on the output value, subject is assigned to a partition [56].This causes imbalance among partitions. This imbalance is shown in example given in Figure 12. Using this technique, our example dataset is split such that, triples 3,10 and 11 are assigned into red partition, triple 7 is assigned into blue partition, and the remaining triples are assigned into green partition. It is a clear imbalance of partitioning.

- **Predicate-hashed.** This technique applies function is applied on the predicate of the triple and based on the output value, predicate is assigned to a partition. This causes all the triples with the same predicate assigned to one partition. In example given in Figure 12, there are four distinct predicate while the required number of partitions are 3. Thus by using the first come for serve strategy, all the triples with predicate $p1$ are assigned to first partition (red), $p2$ triples are assigned to a second partition (green), $p3$ triples are assigned to third partition (blue), and $p4$ triples are again assigned to first partition. This technique can leads to significant performance improvement, provided that the predicates are intelligently grouped intro partitions, such that communication load among data nodes is reduced [4].

- **URI Hierarchy-hashed:** This partitioning technique is based on assumptions of that IRIs have path hierarchy and those with a same hierarchy prefix are often queried together [56]. Same assumptions works in this technique, which extracts the path hierarchy of IRIs and those with same are assigned to same partition. For example in the case of "http://www.w3.org/1999/02/22-rdf-syntax-ns#type", the path hierarchy is "org/w3/www/1999/02/22-rdf-syntax-ns/type". int the subsequent steps, for each level in the path hierarchy (e. g., “org", “org/w3", “org/w3/www", ...), this technique computes the percentage of triples with a same hierarchy prefix. In case of exceeding a percentage to experimentally defined threshold and the number of prefixes is equal to or greater than the number of required partitions at any level of hierarchy, then these prefixes are used for the hash-based partitioning on prefixes. For the determination of IRI prefixes to apply hash function, this technique is computationally expensive. This technique is exhibited in example given in Figure 12, shows all the triples having `hierarchy1` in subjects are assigned to the green partition, triples having `hierarchy2` in subjects are assigned to the red partition, and triples having `hierarchy3` in subjects are assigned to the blue partition. This partitioning may not produce the best query runtimes as the underlying assumptions about IRIs might not be true in practice [4].

Graph-Based Partitioning: It makes use of the graph-based clustering techniques to split a given graph into the required pairwise disjoint sub-graphs. There are three techniques used in this category:

- **Recursive-Bisection Partitioning.** This technique is used to solve the k-way partitioning problem as presented in [60]. It works in three steps i.e Coarsening, Partitioning and Uncoarsening. In *Coarsening:* smaller graphs are generated by the input graph. The partitioning phase computes the 2-way partition which divides the vertices into two parts. In the last phase of Uncoarsening, partitions of a graph are projected back to an original graph.

  This technique is shown by the example given in Figure 12, triples 1, 2, 4, 7, and 8 are labelled green partition, triples 3, 5, 6, 9 and 10 are labelled with red partition, and triple 11 is labelled with blue partition.

- **TCV-Min Partitioning.** This technique is also used to solve the problem of k-way graph partitioning with the main objective is to minimize the *total communication volume* [17] of the partitioning. This technique also comes with phases. But the second phase, i.e. the *Partitioning*, is about the minimization of communication costs. This is shown by example given in Figure 12, triples 1, 2, 4, 5, 6, 8 and 9 are labelled with the green partition, triples 3, 7 and 10 are labelled with the red partition, and triple 11 is labelled with the blue partition.

- **Min-Edgecut Partitioning.** The Min-Edgecut [60] is also used to solve the problem of k-way graph partitioning. However, unlike TCV-Min, the focus is to

[16]We used different example to show a clear difference between the discussed RDF partitioning techniques.

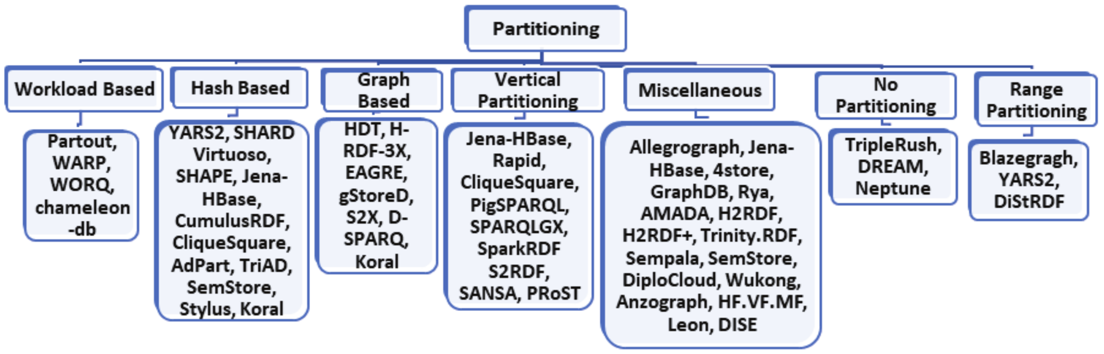

Partitioning
Workload Based
Partout, WARP, WORQ, chameleon -db
Hash Based
YARS2, SHARD Virtuoso, SHAPE, Jena-HBase, CumulusRDF, CliqueSquare, AdPart, TriAD, SemStore, Stylus, Koral
Graph Based
HDT, H-RDF-3X, EAGRE, gStoreD, S2X, D-SPARQ, Koral
Vertical Partitioning
Jena-HBase, Rapid, CliqueSquare, PigSPARQL, SPARQLGX, SparkRDF S2RDF, SANSA, PRoST
Miscellaneous
Allegrograph, Jena-HBase, 4store, GraphDB, Rya, AMADA, H2RDF, H2RDF+, Trinity.RDF, Sempala, SemStore, DiploCloud, Wukong, Anzograph, HF.VF.MF, Leon, DISE
No Partitioning
TripleRush, DREAM, Neptune
Range Partitioning
Blazegragh, YARS2, DiStRDF

Figure 11. Types of Partitioning used in RDF Engines

partition the vertices by minimizing the number of edges connected to them. This is shown by example given in Figure 12, triples 1, 2, 4, 7 and 8 are labelled with green the partition, triples 3, 5, 6, 9 and 10 are labelled with the red partition, and only triple 11 is labelled with the blue partition.

The graph-based partitioning techniques are computationally complex, and may takes very strong resources for splitting big RDF datasets.
Vertical Partitioning: The vertical partitioning is already discussed in section 4. This technique generally divides the given RDF dataset based on predicates[17]. It creates n number of two columns tables, where n is the number of distinct predicates in the dataset. Please note that it breaks the triples and only store the subject and objects parts. The predicate become the caption of the table. Ideally, the number of distinct partitions would be equal to the number of distinct predicates used in the dataset. However, it is possible that the the required number of partitions may be smaller than the number of predicates used in the dataset. In this case the predicate tables are grouped into partitions, i.e., multiple predicate tables are stored in partitions. There can be multiple way of grouping predicates into partitions: (1) first come, first serve, (2) by looking at the number of triples per predicate and thus group predicates such that maximum load balancing is achieved among partitions, (3) using some intelligence to determine which predicates will be queried together, and hence grouped their corresponding triples in one partition.

9. STATE-OF-THE-ART RDF ENGINES

Now we present state-of-the-art centralized and distributed RDF Engines. The goal is to provide a broader overview of these engines and classify them according to the previously discussed data storage and partitioning, indexing, language, and query processing techniques. Summary of these characteristics in centralized and distributed RDF engines is shown in table 9 and 10.

9.1 Centralized RDF Engines

[17]Division by subject and object is also possible but not common in RDF partitioning.

Redland [12] is a set of RDF libraries for storing and querying RDF data that can be used by other RDF-based applications. It provides a TT like storage based on creating three hashes – SP2O, PO2S, SO2P – per RDF triple, where S, P, O stands for Subject, Predicate and Object respectively. Each hash is a map of a key to a value with duplicates allowed. The first two characters represent the hash key and the last character represent the value. For example, in SP2O the key for this hash is subject and predicate and value is the object. These hashes also serve as three triple-permutation indexes. The hashes can be stored either in-memory or on a persistent storage. It creates an RDF model which can be queried with SPARQL and RDQL using the Rasqal RDF query library[22].

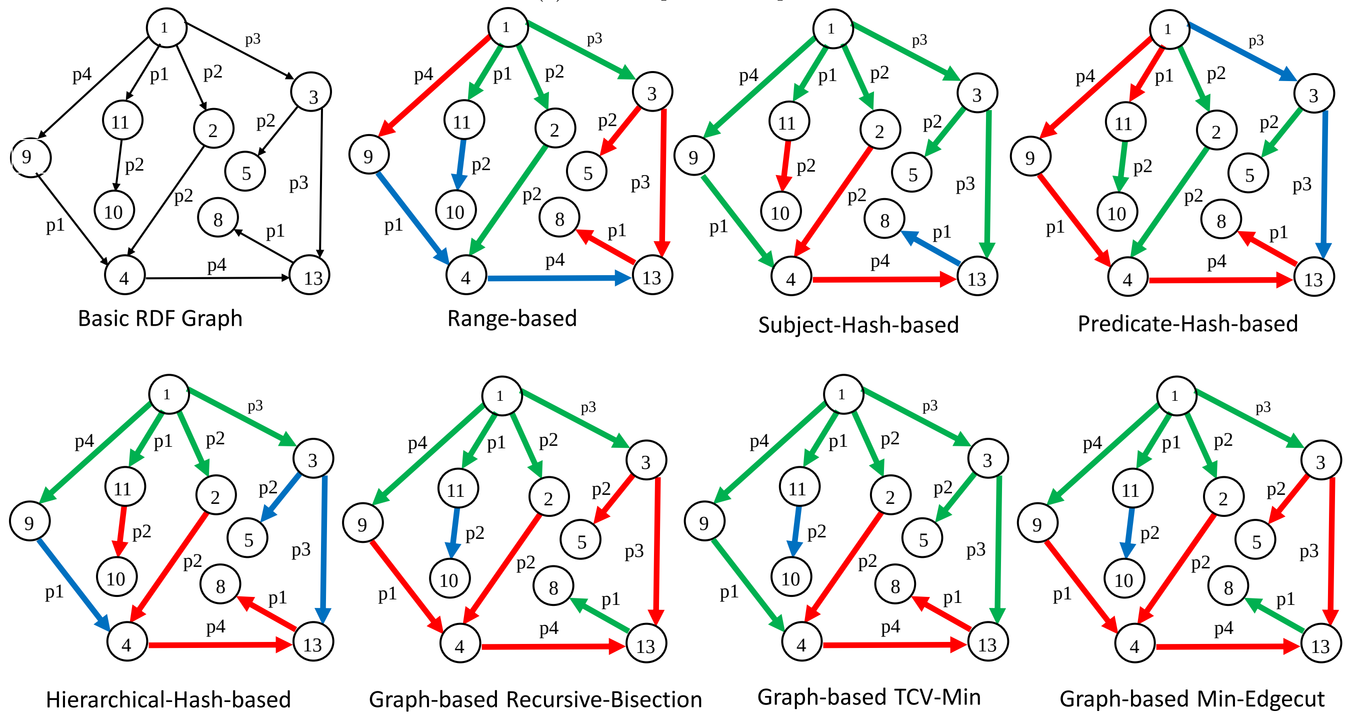
Jena1 [72] uses relational databases to store data as TT called statements tables. Jena1 statement table has entries for subject, predicate, objectURI, and objectliteral. URIs and Strings are encoded as IDs and two separate dictionaries are created for literals and resources/URIs. This scheme is very efficient in terms of storage because of the one-time storage of multiple occurrences of URIs and literals. However, the query execution performance is greatly affected by the multiple self joins as well as dictionary lookups. The indexing, query processing depends upon the relational DBMS (e.g., Postgresql, MySQL, Oracle) it uses for storage. RDQL is used as a query language that is translated into SQL to be run against the underlying relational DBMS.

TDB[23] is a component of the Jena API[24] for storing and querying RDF data. It runs on the single machine and supports full Jena APIs. A dataset in TDB consists of three table namely the node table, the triple and Quad indexes, and the prefixes table. TDB assigns a node ID to each dataset node and is stored in a dictionary table called node table. Triple and quad indexes table uses: (1) three columns or TT to store all the RDF triples belonging to the default named graph, (2) four columns or quads to store triples (along with the corresponding named graphs), belonging to other named graphs. Prefixes table does not take part in a query processing and only contains node table and an index for GPU. For query processing, TDB makes use of

[22]RASQAL library `http://librdf.org/rasqal/`
[23]https://jena.apache.org/documentation/tdb/architecture.html
[24]Jena: `https://jena.apache.org/`

```
@prefix hierarchy1: <http://first/r/> . @prefix hierarchy2: <http://second/r/> .
@prefix hierarchy3: <http://third/r/> . @prefix schema: <http://schema/> .
hierarchy1:s1     schema:p1     hierarchy2:s11 .  #Triple 1
hierarchy1:s1     schema:p2     hierarchy2:s2 .   #Triple 2
hierarchy2:s2     schema:p2     hierarchy2:s4 .   #Triple 3
hierarchy1:s1     schema:p3     hierarchy3:s3 .   #Triple 4
hierarchy3:s3     schema:p2     hierarchy1:s5 .   #Triple 5
hierarchy3:s3     schema:p3     hierarchy2:s13 .  #Triple 6
hierarchy2:s13    schema:p1     hierarchy2:s8 .   #Triple 7
hierarchy1:s1     schema:p4     hierarchy3:s9 .   #Triple 8
hierarchy3:s9     schema:p1     hierarchy2:s4 .   #Triple 9
hierarchy2:s4     schema:p4     hierarchy2:s13 .  #Triple 10
hierarchy2:s11    schema:p2     hierarchy1:s10 .  #Triple 11
```

(a) An example RDF triples



(b) Graph representation and partitioning. Only node numbers are shown for simplicity.

Figure 12: Partitioning an example RDF into three partitions using different partitioning techniques. Partitions are highlighted in different colors.

Table 9: Categorization of centralized RDF Engines.
Storage (T = Triple Table, P = Property Table, V = Vertical Partitioning, G = Graph-based, M = Miscellaneous)
Indexing (P = Predicate-based, T = Triple-permutation, B = Backend Database, N = No-indexing, M = Miscellaneous)
Language (S = SPARQL, T = SPARQL Translation, O = Others Languages)
Query Processing (S = Statistics-based, T = Translation-based, A = API-based, G = Subgraph Matching-based, M = Miscellaneous)

	Storage					Indexing					Language			Query Processing				
Engine	T	P	V	G	M	P	T	B	N	M	S	T	O	S	T	A	G	M
RedLand [12]	✓						✓				✓		✓			✓		
Jena1 [72]	✓							✓					✓					✓
TDB[18]	✓						✓				✓			✓				
Sesame [16]					✓			✓					✓			✓		
3store [42]	✓							✓					✓		✓			
Jena2 [121]	✓	✓						✓					✓					✓
Mulgara[19]	✓						✓						✓	✓				
RStar [69]	✓							✓					✓					✓
BRAHMS [53]	✓						✓						✓					✓
YARS [45]	✓						✓						✓	✓				
Kowari [122]	✓						✓						✓	✓				
RDF-Match [20]	✓							✓					✓					✓
SW-Store [2]			✓			✓						✓						✓
Hexastore [118]					✓		✓				✓							✓
RDFJoin[20]	✓						✓				✓							✓
Parliament [64]	✓									✓			✓					✓
DOGMA [15]				✓						✓			✓				✓	
Turbo$_{HOM++}$ [63]				✓		✓					✓						✓	
RDF-3X [78]	✓						✓				✓			✓				
RIQ [61]					✓					✓	✓							✓
Stardog[21]					✓		✓				✓							✓
gStore [133]				✓			✓				✓						✓	
Strabon [66]					✓					✓			✓	✓				
BitMat [10]					✓					✓	✓							✓
Triplebit [129]					✓					✓	✓							✓
DB2RDF [14]		✓							✓			✓			✓			
RDFox [76]					✓					✓	✓							✓
AMBER [51]				✓						✓	✓						✓	

the `OpExecutor` extension point of the Jena ARQ[25]. TDB provides low level optimization of the SPARQL BGPs using a statistics based optimizer. FUSEKI[26] component of the Jena can be used to provide a public http SPARQL endpoint on top of TDB data storage.

Sesame [16] is an architecture that allows persistent storage and querying of RDF data. Sesame provides storage-independent solutions and hence it can be deployed on top of a variety of storage devices such as relational DBMS and Object-oriented databases. The querying is based on RQL language. The storage, indexing, and query processing is based on the underlying DBMS used by the Sesame. Sesame has been renamed as Eclipse RDF4J[27] with an improved functionalities such as both in-memory and persistence data storage, SPARQL and SeRQL support etc.

3store [42] uses MySQL[28] as its back end and arranges its data in MySQL database schema in four tables namely *triples table*, *models table*, *resource table*, and *literal table*. The *triples table* stores RDF triples (one per row) with additional information: (1)the model this triple belongs, (2) the Boolean value if a literal is used in the triple, and (3) and the Boolean value if this triple is inferred. The *models*, *resource*, and *literal* are two columns tables to map hash IDs to models, resources, and literals, respectively. As **3store** uses MySQL as its backend, it depends upon the MySQL built-in query optimizer to effectively use its native indexes. The query processing is based on the translating RDQL to SQL.

Jena2 [121] is an improved version of the Jena1. It has support both for statement and property tables. Unlike Jena1, the schema is denormalized and URIs, simple literals are directly stored within the the statement table. Jena2 also makes use of the dictionary tables only for strings whose lengths exceed a threshold. The advantage of directly storing URIs and literals into TT is that the filters operation can be directly performed on TT, thus avoiding extra dictionary lookups. The disadvantage is that it also results in higher storage consumption, since string values are stored multiple times. The use of property tables for most frequently used predicates can avoid the multiple self joins in statement tables.

[25]Jena ARQ: `https://jena.apache.org/documentation/query/`
[26]FUSEKI: `https://jena.apache.org/documentation/fuseki2/`
[27]RDF4J: `https://en.wikipedia.org/wiki/RDF4J`
[28]https://www.mysql.com/

Mulgara[29] uses transactional triple store XA[30] as its storage engine. It stores metadata in the form of subject-predicate-object. It makes use of the triple-permutation indexing based on **S**ubject, **P**redicate, **O**bject and Meta or **M**odel. A total of six indexes[31] – SPOM, POSM, OSPM, MSPO, MPOS, MOSP – are used in Mulgara which are stored in AVL (Adelson-Velskii and Landis) (AVL) trees. iTQL is the language used for querying. The query planning is based on cardinality estimations.

RStar [69] was designed to store ontology information and instance data. The storage is based on multiple relations, stored in relational IBM DB2 DBMS. Five two-column tables were used to store Ontology-related data. Similarly, another five two-columns tables were used to store instance-related data. RStar Query Language (RSQL) was defined for resource retrieval. RStar performs translation of RSQL into the SQL of an underlying database. Query engine of RSQL pushes many tasks to the underlying database to take advantage of built in query evaluation.

BRAHMS [53] main memory-based RDF engine. The RDF data is store in three hash tables s-op, o-sp, p-so. This RDF engines was designed to find semantic associations in large RDF datasets. This was done by leveraging the depth-first search and breath-first search algorithms.

Yet Another RDF Store (YARS) [45] stores RDF data in the form of quads having four columns in the disk. In addition to subject, predicate, object, it also stores the context of the triple. YARS implements two kinds of indexes: (1) Lexicons indexes, which operate on the string representations of RDF graph nodes to enable fast retrieval of object identifiers, (2) Quad indexes, which are a set of permutation indexes applied on quad (subject s, predicate p, object o and context c). The are six – SPOC, POC, OCS, CSP, CP, OS – quad indexes used in the YARS. YARS uses Notation3 (N3) to query RDF data. The query processing is based on index lookups and join operations. The join ordering (lowest cardinality join should be executed first) is based on getting the actual cardinalities using `getCountQ` query.

Kowari [122] is a metastore built to provide scalable, secure transactions, and storage infrastructure for RDF data. Kowari makes use of the persistent quad–storage by using *XA Statement Store*. Along with subject, predicate, object, it also stores additional meta node with each triple. The meta node illustrates in which model this statement occurs. Multiple quad-permutation indexes are implemented. Same like Mulgara, Kowari also implements six indexes which are stored as an AVL tree and B-Trees. This combination enables fast, simple searching and modification. The query processing is based on iTQL. Kowari query execution plan is also based on cardinality estimations. Once the initial execution plan is complete, the query engine starts to merge the results using join operations. During this process, the query engine may find a more efficient alternate execution plan from the sizes of intermediate results. If this happens, the query engine will further optimize the execution plan, and complete results are returned to a client after the final join operation.

RDF-Match [20] is implemented on top of Oracle RDBMS using Oracle's table function infrastructure. It stores RDF data in two different tables: (1) `IdTriples` table consisting of columns ModelID, SubjectID, PropertyID, ObjectID, and (2) `UriMap` table consisting of UriID, UriValue. It translates the query into a self-join query on `IdTriples` table. It defines B-tree indexes and materialized views on both tables. SQL is used as a query language to perform queries. For efficient query processing, materialized join views and indexes are used. RDF-MATCH also uses a Kernel enhancement to eliminate runtime overheads.

SW-Store [2] is the example of vertical partitioning: the data is divided into n two columns (subject, object) tables, where n is the number of distinct predicates in the dataset. In addition to natural indexing by predicate, each of the n tables is indexed by subject so that particular subjects can be retrieved quickly from the tables. The implementation of SW-Store relies on a column-oriented database system C-store [111]. SW-Store uses Jena ARQ to translate SPARQL queries into SQL. For triple patterns with bound predicates, only one table is consulted to get the required results. A fast merge-join operations are exploited to collect information about multiple properties for subsets of subjects.

Hexastore [118] proposes an RDF storage scheme that uses the triple nature of RDF as an asset. In this approach, a single giant TT is indexed in six possible ways – SPO, sop, pso, pos, osp, ops. In this storage technique, two vectors are associated (ref. Figure 8) with each distinct instance of the subject, predicate or object. This format of extensive indexing allows fast query execution at the price of a worst-case five-fold increase in index space as compared to single TT. Hexastore uses SPARQL as its query language. The query processing is based on using the appreciate index for the different types of triple patterns (ref. Figure 6). Since, all vectors store elements in sorted order, a fast merge-joins can be used to integrate the results of the different triple patterns.

RDFJoin[32] contains three types of tables namely the `URI conversion tables`, `TT`, and `join table`. The `URI conversion tables` are just like dictionary encoded values of the URIs. RDFJoin stores triple in three TTs namely `PSTable`, `POTable`, and `SOTable`. The `PSTable` consists of columns for PropertyID, SubjectID and ObjectBitVector. The PropertyID and SubjectID represent the property resp. subject of a triple and ObjectBitVector is a bit vector of all objects that are associated with given property and subject instances. The same explanation goes for `POTable` and `SOTable`. The `PSTable` is naturally ordered by the property and secondary indexed by the subject. Both property and subject make the primary key of the table which enables to find out Object by the primary key lookups. The same explanation goes for `POTable` that facilitates the merge joins in the case of Subject-Subject and Object-Object joins. The `Join tables` store the results of subject-subject joins, object-object joins, and subject-object joins in a bit vectors. Three separate join tables namely `SSJoinTable`, `SOJoinTable`, and `OOJoinTable` are used to store the subject-subjec, object-object, and subject-object joins, respectively. Each of the three join tables has three columns for Property1, Property2 and a BitVector. The first two columns make the primary key upon which hash indexed is applied to produce a corresponding BitVector. The query execution is based on SPARQL query processing with index lookups and different join implementations such as merge

[29]http://mulgara.org/

[30]XA1: `https://code.mulgara.org/projects/mulgara/wiki/XA1Structure`

[31]https://code.mulgara.org/projects/mulgara/wiki/Indexing

[32]http://www.utdallas.edu/ jpm083000/rdfjoin.pdf.

joins.

Parliament [64] contains three types of tables, i.e. resource table, statement table, and resource dictionary. These tables are stored as linked lists. Most important is the statement table, which stores records pertaining to the RDF triples. Records are stored sequentially with a number as IDs. Each record has seven components: three IDs representing Subject, Predicate, and Object, three IDs for the other triples which are re-using the same Subject, Predicate, and Object instances. The seventh component is a bit field for encoding attributes of a statement. The index structure of Parliament revolves around the Resource table and a memory-mapped file containing the string representations of Resources ID. Records or instances can be accessed through a simple array indexing technique by giving its ID. Parliament is an embedded triple store, so directly querying through SPARQL or any other query language is not possible. Rather, it allows search, insert, and delete operations. However, querying can be indirectly made possible while accessing it through some SPARQL API like Jena and Sesame. For example, the given SPARQL SELECT query needs to be converted in the format of Parlimaent's supported find query (or set of queries) to be executed.

DOGMA [15] presents a graph-based RDF data storage solution. In this model, RDF graph is represented as balanced binary tree and store it on disk. There is no specific index needed for query processing as subgraph matching; the tree itself can be regarded as index. However, author have proposed two additional indexes for fast subgraph matching. DOGMA develops algorithms to answer only graph matching queries expressible in SPARQL, and hence it was not supporting all SPARQL queries.

Turbo$_{HOM++}$ [63] is another graph-based solution for running SPARQL queries. However, unlike DOGMA, it is an in-memory solution in which both RDF graph and SPARQL query is represented as specialized graphs by using type-aware transformation, i.e., it makes use of the type information specified by the `rdf:type` predicate. The subgraph matching is performed for SPARQL query execution. This approach also makes use of the predicate index where a key is a predicate, and a value is a pair of a list of subject IDs and a list of object IDs.

RDF-3X [78] is an example of exhaustive indexing over a single giant TT by using optimized data structures. Triples are stored in a clustered B+ trees in lexicographic order. The values inside the B+ tree are delta encoded [78] to further reduce the space required for storing these indexes. A triple $< S, P, O >$ is indexed in six possible ways – spo, sop, pso, pos, osp, ops –, one for each possible ordering of the subject s, predicate p, and object o. These indexes are called *compressed indexes* in RDF-3X. In addition, RDF-3X makes use of the six *aggregated indexes* – (sp, ps, so, os, po, op – each of which stores only two out of the three components of a triple along with an aggregated count which is the number of occurrences of this pair in the full set of triples. The aggregated indexes (value1, value2, count) are helpful for answering SPARQL query of type "$select?a?cwhere?a?b?c$ ". Finally, three one value indexes (value1, count) are also maintained for storing the count of the s, p, and o. Thus, all together 15 indexes are created in RDF-3X. SPARQL query processing and the selection of optimized query execution plan is based on a cost model which calculates the cost of the different plans and select the plan with minimum cost.

RIQ [61] is based on the storage of RDF data as new vector representation. This is achieved by applying different transformations related to a triple in RDF graph and a triple pattern in a BGP. RIQ also maps a graph with context c to pattern vector (PV). This is done by hash function based on a Rabin finger printing technique [88]. New vector representation helps to groups same RDF graph. Inspite of whole RDF graph, these similar groups are indexed in RIQ. A novel filtering index called PV-Index is used for this purpose. This index is used to identify candidate RDF graph early. It is a combination of a Bloom filter and a Counting Bloom Filter for compact storage. RIQ employs a divide and conquer approach for SPARQL query processing. This approach works on the identification of early groups by PV-Index. After the identification different optimizations are applied and SPARQL query is rewritten and executed on the SPARQL processor which support quads to represent final output.

Stardog[33] storage is based on the RocksDb[34], a persistant key-value store for fast storage. It supports the triple-permutation indexes where triples are represented as quads, thus the fourth element of the quad (i.e, the context) is also indexed. It supports SPARQL 1.1, full-text search through Lucene and ACID transactions. The query planning is based on index search with support for different types of joins such as hash join, bind join, merge join etc.

gStore [133] is a graph-based engine that stores RDF triples in the form of directed, multi-edge graph. The subject and object of a triple is represented by graph nodes and the corresponding predicate represents a directed link from subject node to object node. Multi edges between two nodes can be formed if there exist more than one property relation between the subject and object. The RDF graph is stored in an adjacency list table and encoded into a bitstring, also called vertex signature. The bitstring encoding of the graph is been done by using different hash functions [23]. gStore uses two trees for indexing: (1) a height-balanced S-tree [26], and (2) VS-tree (vertex signature tree). The S-tree index can be used to find attribute values in the adjacency list specified in a query. But it is not able to support multi-way joins over attribute values. To solve this problem, gStore makes use of the VS-tree. Same like graph encoding, a SPARQL query is also represented as signature graph called *query signature*. Consequently, query execution problem is reduce to the sub-graph matching problem where query signature graph is matched against the graph signature.

Strabon [66] is used for storing and querying geospatial data, which exists in the form of stRDF, originally presented in [65]. Strabon is built upon Sesame also known as RDF4J. Sesame is chosen because it is an open-source nature, consists of layered architecture, offers a wide range of functionalities and it has the ability to have PostGIS, which is a database management system with spatial features. Strabon has three components which consists of the storage manager, the query engine and PostGIS. The storage manager of Strabon makes use of a bulk loader to store stRDF triples. These triple are stored in a storage scheme of Sesame, "one table per predicate" and dictionary encoding. For each of predicate and dictionary tables, Strabon creates a two B+ tree two-column indexes and a B+ tree index on the id column re-

[33]https://www.stardog.com/docs/7.0.0/
[34]RocksDB: `https://rocksdb.org/`

spectively. Strabon has new extension of SPARQL query language, stSPARQL [65] to query stRDF based datasets.

The query engine of Strabon performs the query execution. It consists of a parser, an optimizer, an evaluator and a transaction manager. The parser and the transaction manager are same like in a Sesame. An optimizer and an elevator are modified in Strabon. The parser generates an abstract syntax tree, which becomes a query tree by mapping into the Sesame's algebra. The the optimizer takes the query tree and applies different optimizations and then an evaluator produces the corresponding SQL query. This SQL query will be executed on PostgreSQL.

BitMat [10] as in example of binary data storage which makes use of the three-dimensional (subject, predicate, object) bit matrix which is flattened to two dimensions for representing RDF triples. In this matrix, all the values used are either 0 or 1, representing the absence or presence of that triple. BitMat further compress the data on each row level. In particular, BitMat creates three auxiliary tables to get mappings of all distinct subjects, predicates, and objects to the sequence-based identifiers. Bitwise AND/OR operators are used to process join queries.

Triplebit [129] stores RDF data in the form of bit matrix storage structure [10] and applies an encoding mechanism to compress huge RDF graphs. The RDF triples are represented as a two-dimensional bit matrix. The columns of the matrix represents triples, with only bit value entries for subject and object of the triple. Each row is defined by a distinct entity value which represents set of triples which contain this entity. TripleBit vertically partitions the matrix into multiple disjoint buckets, one bucket per predicate. Two indexes are used in TripleBit namely `ID-Chunk bit matrix` and `ID-predicate bit matrix`. The former supports a fast search for finding the relevant chunks for given subject or object. The later provides a mapping of a subject or an object to the list of corresponding predicates to which it relates. A dynamic query planning algorithm is used to generate the optimized SPARQL query execution plans, with the aim of minimizing the size of intermediate results as early as possible. Merge joins are used extensively in the generated optimized query execution plans.

DB2RDF [14] uses a relational schema consisting property tables to store RDF data. The storage is based on the encoding scheme which encodes a list of properties for each subject in a single row. DB2RDF does not create any specific index over triple $< s, p, o >$. However, the property tables can be naturally regarded as subject based index to quickly locate all the properties and the corresponding objects for the given subject instance. DB2RDF performs query optimization in two steps. In the first step SPARQL is optimized, and in the second step SPARQL to SQL translation is optimized.

RDFox [76] is an in-memory RDF engine that supports materialisation-based parallel datalog reasoning and SPARQL query processing. The RDF triples are stored in TT. Three different types of indexes are created over the TT namely `ThreeKeysIndex`, `TwoKeysIndex`, and `OneKeyIndex`. These indexes are used to efficiently answer the different types of triple patterns (ref. Figure 6). For example, the `ThreeKeysIndex` can be used for answering triple patterns containing bound subjects, predicates, and objects (i.e., variable-free patterns). The `TwoKeysIndex` can be used for answering triple patterns containing one or two variable. Each `TwoKeysIndex` contains a `OneKeyIndex` that is used to locates the first triple in the relevant list with the given resource ID. RDFox answers SPARQL queries by using its querying package, which first parse the SPARQL query int a query object. Then, the SPARQL compiler converts the query object into a `TupleIterator` that provides iteration over the answers. RDFox contains many different `TupleIterator` variants such `TableIterator` supports iteration over SPARQL triple patterns, `DistinctIterator` implements the "DISTINCT" construct of SPARQL etc.

AMBER (Attributed Multigraph Based Engine for RDF querying) [51] stores all of its RDF data in the form of multigraph. It creates three different dictionaries of key value pairs namely a vertex dictionary, an edge-type dictionary and attribute dictionary. Same like three dictionaries, AMBER creates three indexes: first is an inverted list to store the set of data vertex for each attribute, second is a trie index for storing features of the data vertices, and third is also a trie index structure for storing information of neighbours of the data vertices. SPARQL query in **AMBER** is transformed into a query mutligraph and then applied on the data multigraph. Results are obtained by using the subgraph matching mechanism of the query and data multigraphs.

9.2 Distributed RDF Engines

The distributed RDF engines can be divided into four broader categories : (1) No-SQL-based, (2) Hadoop/Spark-based, (3) Distributed memory-based, and (4) others, e.g., MPI-based, Graph-based. Our discussion is focused towards the storage, indexing, query processing, and partitioning in state-of-the-art distributed RDF engines.

AllegroGraph[40] can be used to store and query both documents (e.g. JSON) and graph data (e.g. RDF). The data is horizontally distributed also called shards. It uses efficient memory management in combination with disk-based storage, enabling it to scale up to billions of quads. For RDF data, it makes use of the triple-permutation indexes along with named graph and triple id. The default indexes are: are: spogi, posgi, ospgi, gspoi, gposi, gospi and i, where s is the subject, p is the predicate, o is the object of a triple pattern, g is the named graph and i is the unique ID of the triple. Query processing is based on two components [41]. The first is a simple static query analyzer, which indicates the best indexes to be used during query processing. The second one is a dynamic query analyzer, which processes the query and determines which indexes are actually used. There is a trade-off in the use of the dynamic query analyzer: by providing better information, it takes a much longer time processing the query.

Blazegraph[42], formerly known as *BigData*, is an open-source graph database system which supports both RDF and SPARQL. Blazegraph supports both row and column data storage models, which can be saved both in-memory or on a disk. Indexing is dependent on the triples or quads or triples with provenance. Based on the type of triple Bigdata creates three or six key-range partitioned B+ tree indexes. Indexing in Bigdata is like in YARS. Dictionary encoding with 64bit integers is used for compressed representation of RDF triples. The query optimizer of Bigdata generates optimized query execution plans according to two different approaches: the

[40]https://franz.com/agraph/allegrograph/
[41]https://franz.com/agraph/support/documentation/current/query-analysis.html
[42]https://blazegraph.com/

Table 10: Categorization of distributed RDF Engines.
Storage (T = Triple Table, P = Property Table, V = Vertical Partitioning, G = Graph-based, M = Miscellaneous)
Indexing (P = Predicate-based, T = Triple-permutation, B = Backend Database, N = No-indexing, M = Miscellaneous)
Language (S = SPARQL, T = SPARQL Translation, O = Others Languages)
Query Processing (S = Statistics-based, T = Translation-based, A = API-Based, G = Subgraph Matching-based, M = Miscellaneous)
Partitioning (W = Workload-based, H = Hash-based, G = Graph-based, V = Vertical, M = Miscellaneous, N = No partitioning, R= Range Partitioning)

	Storage					Indexing					Language			Query Processing					Partitioning						
Engine	T	P	V	G	M	P	T	B	N	M	S	T	O	S	T	A	G	M	W	H	G	V	M	N	R
Allegrograph[35]					✓		✓				✓							✓					✓		
Blazegraph [36]					✓		✓				✓			✓											✓
YARS2 [46]					✓		✓				✓							✓		✓					✓
SHARD [90]					✓				✓		✓							✓		✓					
4store [43]	✓						✓				✓			✓									✓		
Virtuoso [30]	✓						✓					✓			✓					✓					
HDT [32]				✓			✓				✓							✓			✓				
H-RDF-3X [50]	✓						✓				✓			✓				✓			✓				
GraphDB [37]					✓					✓	✓							✓					✓		
CumulusRDF [44]					✓		✓				✓					✓				✓					
Rapid+ [89]			✓							✓		✓			✓							✓			
Jena-HBase [62]	✓		✓		✓		✓				✓							✓		✓		✓	✓		
Rya [86]					✓		✓				✓			✓		✓							✓		
AMADA [9]					✓		✓				✓		✓					✓					✓		
H2RDF [83]					✓		✓				✓			✓									✓		
SHAPE [67]	✓						✓				✓			✓				✓		✓					
WARP [49]	✓						✓				✓							✓	✓						
PigSPARQL [102]			✓						✓			✓			✓							✓			
EAGRE [132]					✓					✓	✓							✓			✓				
H2RDF+ [82]					✓					✓	✓			✓									✓		
Trinity.RDF [131]				✓			✓				✓						✓						✓		
D-SPARQ [75]				✓			✓				✓			✓							✓				
CliqueSquare [34]			✓						✓		✓						✓			✓		✓			
TripleRush [113]				✓						✓	✓						✓							✓	
chameleon-db [8]				✓						✓	✓						✓		✓						
Partout [33]	✓						✓				✓			✓					✓						
Sempala [103]		✓							✓			✓			✓								✓		
TriAD [39]				✓			✓				✓			✓						✓					
SparkRDF [19]				✓						✓	✓					✓	✓					✓			
SemStore [123]				✓						✓	✓			✓						✓			✓		
DREAM [40]	✓						✓				✓			✓										✓	
DiploCloud [125]					✓					✓	✓			✓									✓		
SPARQLGX [102]			✓			✓						✓			✓							✓			
S2RDF [104]			✓			✓						✓		✓								✓			
AdPart [41]					✓		✓				✓			✓						✓					
S2X [101]				✓					✓		✓						✓				✓				
gStoreD [85]					✓				✓		✓						✓				✓				
Wukong [108]				✓						✓	✓							✓					✓		
SANSA [68]			✓			✓						✓			✓							✓			
Stylus [48]					✓					✓	✓						✓			✓					
Koral [55]				✓			✓				✓		✓					✓		✓	✓				
PRoST [24]			✓			✓					✓			✓								✓			
WORQ [71]			✓							✓	✓							✓	✓						
Anzograph[38]				✓					✓		✓							✓					✓		
Neptune[39]				✓			✓				✓		✓	✓										✓	
HF,VF,MF [84]					✓				✓		✓						✓						✓		
DiStRDF [119]	✓	✓								✓	✓							✓							✓
Leon [38]					✓					✓	✓							✓					✓		
DISE [52]					✓		✓					✓						✓					✓		

default approach uses static analysis and fast cardinality estimation of access paths, the second approach uses runtime sampling of join graphs. In remove the scaling limit, the Blazegraph employs a dynamically partitioned **key-range shards**.

YARS2 (Yet Another RDF Store, Version 2) [46] data storage is based on RDF quads (subject, predicate, object, context). The main focus of YARS2 is on the distributed indexing. The index manager in YARS2 uses three indexes namely keyword index, quad index and join index for evaluating queries. The keyword index is used for keyword lookups. The quad indexes are the triple permutation indexes, applied on YARS2 RDF quads. The join index helps speeding up query execution, containing certain combinations of values, or paths in the graph. YARS2 also uses SPARQL as its query language. YARS2 performs the distributed query execution by using optimized methods for handling the network traffic and avoiding memory overheads, index loop joins, and dynamic programming for joins ordering. Finally, a hash-based data placement used for quads indexing.

SHARD [90] is Hadoop-based distributed RDF engine, which employs hash-based data distribution. It stores RDF data in flat files on HDFS in a way that each line presents all the triples associated with a *subject* resource of the RDF triple. Input dataset is hash partitioned, so that every partition contains the distinct set of triples. There is no specific indexing employed in SHARD. Query execution is performed through MapReduce iterations: first, it collects the results for the sub-queries, following by the joins, and finally filtering is performed for bounded variable and to remove the redundant results.

4store [43] stores data in quad (model, subject, predicate, object), where model represents the graph in which triples are saved. The partitioning of data among n partitions is exactly same like subject-based-hashed partition. However, RIDs (Resource IDentifiers) [120] values of the triple subjects are used instead of using hashing function over the subject component of the triple. The allocation of triples to a specific partition is based on the formula defined as: $n = RID(subject) mod n$. Three different types of indexes are used in 4Store namely R , M, and P indexes. RDF Resources, i.e. URIs, Literals, and Blank Nodes are represented as triple (rid, attr, lexical value), and stored in a bucketed, power-of-two sized hash table, known as the R Index. The M index is created over named graphs or model. It is basically a hash table which maps named graphs to the corresponding triples in the named graph. The P Indexes (two for each predicate) corresponds to the predicates and consist of a set of radix tries [73], using a 4-bit radix. The query engine is largely based on Relational Algebra. The join ordering is based on the cardinality estimations.

Virtuoso [30] stores RDF data as single table with four columns: Subject (s), Predicate (p), Object (o), and Graph (g), where s, p and g are IRIs, and o may be of any data type. The Virtuoso makes use of the dictionary encoding for IRI's and literals used in the triples. It creates a fewer number of indexes. The `gspo` is the default index used in the Virtuoso. In addition, it creates an auxiliary bitmap index `opgs`. Query execution is based on the translation of SPARQL queries into SQL to be executed on the underlying database backend. The optimized query plan generation is based on the cost estimations functions. Virtuoso partitions data using subject-hash-based partitioning.

The Header Dictionary Triples **HDT** [32] is a compressed representation of RDF data, which consists of three components : (1) a Header which contains metadata about the dataset, (2) a Dictionary which encodes strings into integers, and (3) Triples which are encoded in a binary format. For each distinct subject in the dataset, a binary tree is created, which consists of three levels. The subject is the root of the tree, followed by the list of predicates and objects containing the given subject. Thus, HDT data structure is naturally indexed by subject. The triple patterns with bound subjects can be directly answered from the binary tree corresponding to the given bound subjects. Additionally, an object-based index is created to facilitate executing triple patterns with bound objects. The join between triple patterns is performed by using merge and index joins. The HDT framework includes distributed query processing tools using Hadoop as well.

H-RDF-3X [50] is a Hadoop-based distributed RDF engine that uses RDF-3X on a cluster of machines. To reduce the communication cost and speedup the query processing, it also leverages n-hop guarantees data replication at workers nodes. A graph-based partitioning (based on METIS) is used to distribute triples among multiple worker nodes. The query execution is based on the RDF-3X and/or Hadoop, which utilizes the RDF-3X's internal statistics to generate efficient query plans. Please note that the presented architecture is flexible and can be replaced with any other centralized RDF engine.

GraphDB *storage*[43] is packaged as Storage and Inference Layer (SAIL) component for the RDF4J framework[44]. The Sail API is a set of Java interfaces that support RDF storing in different mechanism, e.g., using relational databases, file systems, in-memory storage etc. **GraphDB** contains many automatic indexes apply to implicit or explicit triples. Two main triple-permutation indexes namely Predicate, Object, Subject (POS) and Predicate, Subject, Object (PSO) are created in GraphDB. Furthermore, it also creates an optional PSCO and POCS indexes, where C stands for the context. Finally, it also creates literal index for efficient data storage and lookup. The SPARQL query execution is based on RDF4J framework. The GraphDB EE is a master-slave-based distributed RDF engine, where each cluster has at least one master node that manages one or more worker nodes, each hosting the full database copy.

CumulusRDF [44] works on top of the Apache Cassandra, an open-source nested key-value store. Two data storage layouts namely *hierarchical layout* and *flat layout* are implemented in CumulusRDF. The hierarchical layout is based on Cassandra's supercolumns. The flat layout is based on a standard key-value storage model. Three indexes namely SPO, POS, OSP are created to satisfy all six RDF triple patterns.CumulusRDF also supports a new index CSPO, for RDF named graphs by storing quads [1]. There are no dictionaries created in CumulusRDF. The distribution of data among multiple hash tables is based on hashing RDF triples. For query processing, Sesame processor translates the queries into index lookups of Cassandra.

Rapid+ [89] is a Pig-based system that uses a Vertical Partitioning (VP) strategy for storing RDF data. The *SPLIT* command of Pig is used to store data in VP tables. The main goal of the presented approach is to minimize the communication and I/O costs during the map-reduce operations while processing SPARQL queries on top of RDF in the Map Reduce frameworks. To achieve this goal, an intermediate algebra called the Nested TripleGroup Algebra (NTGA) was introduced in combination with predicate-based indexing structure. The NTGA is used for efficient translation of SPARQL queries into PigLatin operations.

Jena-HBase [62] is an HBase backed RDF engine that can be used with the Jena framework. The Jena-HBase supports a variety of custom-built RDF data storage layouts for HBase namely namely Simple, Vertical Partitioned, Indexed, Hybrid and Hash over several HBase tables. The indexes used in Jena-HBase depend on the storage layout. Experimental evaluation shows the superiority of Hybrid layout, which leverages the advantages of multiple layouts. For querying,

[43]http://graphdb.ontotext.com/
[44]https://rdf4j.org/

Jena-HBase has query runner block that provides APIs for different storage layouts.

Rya [86] employs Accumulo, a key-value and column-oriented store as its storage backend. However, it can also uses any other NoSQL database as its storage component. Rya stores three index permutations namely SPO, POS, and OSP. The query processing is based on the OpenRDF Sesame SAIL API. In particular, index nested loop joins are used for SPARQL query execution using MapReduce fashion. The count of the distinct subjects, predicates, and objects is maintained and used during joins reordering and query optimization. These counts are only updated in case of data distribution changes significantly.

AMADA [9] is implemented on Amazon Web Services (AWS) cloud infrastructure. This infrastructure supports storage, indexing and querying as software as a service (SAAS). The indexes are built using using SimpleDB, and key-value storage solution which supports SQL queries. SimpleDB supports several indexing strategies. One in them is attribute indexing which creates three indexes for three elements of a triple. In AMADA, query is submitted to a query processor module running on EC2. After that, indexes are lookup in SimpleDB to find the answers related to a query. Results are written in a file stored in S3, whose URI is sent back to the user to retrieve the query answers.

H2RDF [83] stores data as multiple indexes over HBase, in combination with Hadoop framework. In H2RDF, three permutations of S,P,O (SPO, POS, and OSP) are created over the HBase tables in the form of key-value pairs. The query planning is based on the selectivity information and cost estimation of joins.

The Semantic Hash Partitioning-Enabled distributed RDF data management system **(SHAPE)** [67] uses RDF-3X to store RDF triples across nodes of a cluster. Distribution of triples is done by using semantic hash partitioning based on the URI hierarchy: triples with the same Subject or Object are identified and are placed in the same partition. Same like RDF-3X, it creates an indexes covering all possible permutations of Subject, Predicate, and Object, which are stored as clustered B+ trees. Distributed query execution planner coordinates the intermediate results from different client machines, which are loaded into HDFS and joined using MapReduce Hadoop joins.

Workload-Aware Replication and Partitioning **(WARP)** [49] also uses RDF-3X to store triples in partitions among a cluster of machines. The distribution of triples among clusters is based on given query workload containing a set of SPARQL queries. To reduce the inter-cluster communication cost, WARP supports n-hop guarantee replication. WARP also uses the underlying RDF-3X indexes. Multi-pass queries (MPQ) consisting of many triple patterns are converted into one pass queries (OPQ). This conversion is based on the replication of triples at the border of partition. For each of one pass queries, query optimizer creates an execution plan made up of left deep joins tree. One pass query is executed in parallel by all the slave machines, and results are combined using merge joins.

PigSPARQL [102] uses VP for storing triples. Due to no indexing, PigSPARQL is well suited for "Extract, Transform, Load" like scenarios. The main component of the PigSPARQL is a translation of SPARQL queries into Pig Latin [79] scripts on Apache Pig. The translation is based on the algebraic representation of SPARQL query expressions. In the first step, an abstract syntax tree is generated from the SPARQL query using the Jena ARQ framework. The abstract syntax tree is then translated into the SPARQL algebra tree. At this point, based on selectivity information, optimization for filters and rearrangement of triple patterns are applied. Then this optimized algebra tree is traversed bottom up to generate PigLatin expressions for every SPARQL algebra operator. After this, these PigLatin expressions are mapped into MapReduce iterations.

Entity ***A***ware ***G***raph comp***RE***ssion technique (**EAGRE**) [132] stores RDF data in HDFS in a (key, value) fashion that preserves both semantic and structural information of the RDF graphs. It employs a graph-based data partitioning using Metis [59]. EAGRE adopts a **space-filling curve** technique, an in-memory index structure which is used to index high dimensional data to efficiently accelerate the evaluation of range and order-sensitive queries. The SPARQL query processing in EAGRE is completed using worker nodes. The I/O cost is minimized by using efficient distributed scheduling approaches.

H2RDF+ [82] uses HBase[45] tables to store RDF data, dictionaries, and indexes. It makes use of the distributed MapReduce processing and HBase indexes. The cost of the different query joins is estimated by using the stored statistics in the indexes. H2RDF+ adaptively chooses for either single- or multi-machine execution based on the estimated join costs. It makes use of the distributed MapReduce based implementations of the multi-way Merge and SortMerge join algorithms.

Trinity.RDF [131] has been implemented on top of Trinity [106], a distributed memory-based key-value storage system. It stores RDF data in native graph from which is partitioned across multiple clusters by applying hashing on nodes. The distributed SPARQL query processing is managed by using graph exploration, assisted by predicates and triple permutation indexes (which are like PSO or POS).

D-SPARQ [75] is a distributed RDF engine implemented on top of MongoDB, a NoSQL document database. D-SPARQ takes RDF triples as an input and makes a graph. The graph is then partitioned among nodes of a cluster such that triples with the same subject are placed in the same partition. In addition, a partial data replication is then applied where some of the triples are replicated across different partitions for parallel execution of queries. Grouping the triples based on the same subject enables D-SPARQ to efficiently execute star patterns queries (i.e., subject-subject joins). D-SPARQ indexes involve subject-predicate (sp) and predicate-object (po) permutations. For query execution, D-SPARQ employs a selectivity of each triple pattern to reorder individual triple pattern in star pattern queries.

CliqueSquare [34] is a Hadoop-based RDF engine used to store and process massive RDF graphs. It stores RDF data in VP tables fashion using semantic hash partitioning. The goal is to minimize the number of MapReduce jobs and the amount of data transfer. In CliqueSquare, the objective of the partitioning is to enable co-located or partitioned joins. In these joins, RDF data is placed in such a manner that the maximum number of joins are evaluated in a map phase of the map-reduce system. In addition, CliqueSquare also maintains three replicas for fast query processing and increased data locality. In order to apply SPARQL queries on

[45]https://hbase.apache.org/

RDF dataset, the CliqueSquare uses a clique-based algorithm, which divides a query variable graph into clique sub-graphs. 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.

TripleRush [113] is based on the Signal/Collect [112], a parallel and distributed graph processing framework. In TripleRush, three types of vertices exit namely the triple vertices, the index vertices, and the query vertices. The triple vertices correspond the RDF triples with each vertex contains subject, predicate, and object of the RDF triple. The index vertices correspond to the SPARQL triple patterns and used for matching triple patterns against the RDF triples. The query vertices coordinate the query execution. In TripleRush, index graph is formed by index and triple vertices. A query execution is initialized when a query vertex is added to a TripleRush graph. Then a query vertex emits a query particle which is routed by the Signal/Collect to index vertex for matching. There is no partitioning performed in this framework.

chameleon-db [8] an example of workload-aware RDF graph partitioning and distributed RDF engine. It stores RDF data as graph and SPARQL query is represented as graph as well, thus the query execution is reduced to subgraph matching problem. The query execution is optimized using workload-aware partitioning indexing technique, which an incremental indexing technique using a decision tree to keep track of the relevant segments of the queries. It also uses a vertex index which is a kind of hash table containing URIs of vertices in a subset of partitions. It also maintains a range index to keep track of the minimum and maximum literal values.

Partout [33] also uses RDF-3X for storing RDF triples which are partitioned by using Workload-aware partitioning technique. The goal of the partitioning to group all RDF triples in a same partition which are likely to be queried together in SPARQL queries. RDF-3X in Partout creates indexes covering all possible permutations of Subject, Predicate, and Object, which are stored as clustered B+ trees. The SPARQL query is issued at a server which generates a suitable query plan according to RDF-3X execution model by utilizes a global statistics file, which contains information about partitions definition, size, and mapping to a client. The centralized execution plan of RDF-3X is transformed into a distributed plan, which is then refined by a distributed cost model to resolve the triples locations in a cluster. The refined query plan is executed by client machines in parallel, and the final results are joined by Binary Merge Union (BMU) operation.

Sempala [103] uses Parquet[46] (a columnar storage format for Hadoop) for storing RDF triples. The Parquet is designed for supporting a single wide table and thus Sempala uses a single Unified Property Table (UPT) for string complete RDF triples. Sempala does not maintain any additional index. The query execution is performed by translating SPARQL queries into SQL which is executed on top of UPT by using Apache Impala.

Triple Asynchronous and Distributed (TriAD) [39] implements a main memory, master-slave, and shared-nothing architecture based on asynchronous Message Passing protocol. It is based on classical master-slave architecture in which query processing is performed by the slave nodes using autonomous and asynchronous messages exchange. It implements a graph summarization approach for data storage and to facilitate join-ahead pruning in a distributed environment. It creates indexes both at master and slave nodes. The summary graph is indexed in the adjacency list format in large in-memory vectors. A hash partitioning on PSO and POS permutations is applied to distribute data among slave nodes. The slave nodes create six in-memory vectors of triples received from the master node. These are divided into two parts namely Subject-based and Object-based. Subject-based are SPO, SOP, PSO while object-based are OSP, OPS, POS. The SPARQL query execution is based on the stored indexes, dynamic programming, and distributed cost model.

SparkRDF [19] is a Spark-based RDF engine which distributes the graph into multi-layer elastic subgraphs (MESG) by extending vertical partitioning. MESGs are based on the classes and relations to reduce search space and memory overhead. SparkRDF creates five indexes, i.e., C, R, CR, RC and CRC, where C stands for Class subgraphs and R denotes Relation subgraphs. These indexes are modeled as Resilient Discreted SubGraphs (RDSGs). For SPARQL query processing, SparkRDF employs Spark APIs and an iterative join operation to minimize the intermediate results during subgraph matching.

SemStore [123] follows a master-slave architecture having one master node and many slave nodes. It uses a centralized RDF engine *Triplebit* as its storage component. The partitioning of data in SemStore is based on creating Rooted Sub-Graphs (RSGs) to effectively localize SPARQL queries in particular shapes, e.g., star, chain, tree, or a cycle that captures the most frequent SPARQL queries. It implements hash function to assign RSGs to slave nodes. In order to reduce redundancy and localize more query types, a k-mean partitioning algorithm is used to assign RSGs to a cluster nodes. After partition, the data partitioner creates a global bitmap index over the vertices and collect the global statistics. The slave nodes builds local indexes and statistics to be used during local join processing. The SPARQL query is submitted to a master node which parses the query and decides the query as local or distributed. In case of a distributed query, query is distributed among slave nodes as subqueries and executed on the bases of local indexes and statistics.

DREAM [40] uses RDF-3X single node RDF engine as its storage component. It replicates an entire dataset on every node with RDF-3X installed. This replication causes storage overhead but avoids inter partition communication among cluster nodes. DREAM creates indexes covering all possible permutations of Subject, Predicate, and Object, stored as clustered B+ trees. In the query execution phase, the given SPARQL query is first represented as directed graph, which is divided into multiple sub-graphs to be distributed among the compute nodes. The results of the query sub-graphs are combined by using hash-based algorithm.

DiploCloud [125] makes use of three data structures namely the the molecule clusters, template lists, and a molecule index. The molecule clusters create RDF subgraphs, co-locating semantically related data to minimize inter-node operations. The template lists are used to store RDF literals in lists. The molecule index is used to index URIs in the molecule clusters. The storage is based on a single node

[46]http://parquet.apache.org

system called *dipLODocus* [127]. DiploCloud is based on the master slave architecture. The master node contains the key index encoding of URIs into IDs, a partition manager and distributed query executor. The worker node contains a type index, local molecule cluster and a molecule index. In SPARQL query execution, the master node distributes the subqueries among slave nodes, which run subqueries and return intermediate results to a master node. It makes use of the different partitioning strategies like Scope-k Molecules, Manual Partitioning and Adaptive Partitioning to distribute data among cluster.

SPARQLGX [36] vertically partitioned RDF triples by predicates and stores in Hadoop Distributed File System (HDFS) [109]. A seperate file is created for each unique predicate in the RDF, thus each file contains only subject and object. It translates SPARQL queries into Scala code which is then directly executed by using Spark.

S2RDF [104] translates SPARQL queries into SQL which is run on of Spark [130]. The storage is based on an extended version of the vertical partitioning with special cares for tuples that do not find a join partner when performing a join between SPARQL triple patterns. It makes use of the selectivity information for query planning and join ordering.

AdPart [41] is a distributed RDF engine that follows a typical Master-Slave architecture and is deployed on a shared-nothing cluster of machines. The master initially partitioned the data by using by hashing on the subjects of triples. The slave stores the corresponding triples using an in-memory data structure. Within the worker node, AdPart primarily hash triples by predicate, leading to a natural predicate-based index (P-index). In addition, each worker maintain predicate-subject index (PS-index) and predicate-object index (PO-index). The SPARQL query processing is done by using cost estimations using stored statistics.

S2X [101] processes SPARQL queries on the top of the Spark component called GraphX [35]. The triples stored among different worker machines after applying hash-based encoding on Subject and Objects through GraphX default partitioner. It converts the RDF graph into a property graph, which is the data model used by GraphX. S2X does not maintain any indexing. For SPARQL query processing, it combines matching of graph patterns with relational style operators to produce solution mappings. Relational operators of SPARQL are implemented through Spark API.

gStoreD [85] is based on the strategy of partitioning the data graph but not decomposing the query. It stores triples by modifying the centralized RDF engine gStore [133], which stores RDF triples in an adjacency list table and encodes them into a bitstring also known as vertex signature. There is no indexing mechanism used in gStoreD. The SPARQL query processing is based on sub-graph matching.

Wukong [108] stores RDF data in a form a directed graph on a distributed key-value store. It maintains two kinds of indexes: a normal vertex index which refers to a subject or object of a triple; and a predicate index which store all subjects and objects labeled with the particular predicate. In order to distribute SPARQL queries among cluster machines, Wukong makes use of the Remote Direct Memory Access (RDMA). Each query is divided into multiple subqueries on the basis of the selectivity and complexity of queries. Wukong employs a graph exploration mechanism with *Full History Pruning* to remove intermediate results during query execution.

Semantic Analytics Stack(Sansa) [68] is a SPARK-based distributed RDF data management system. It makes use of the extended Vertical partitioning (VP) of RDF data stored in HDFS. Since the data is vertical partitioned by using predicates of the triples, SANSA stack maintains a natural predicate-based index. **Sparklify** [110] is used as default query engine for SPARQL-to-SQL translation of SPARQL queries into Spark SQL to be executed on top of Apache Spark SQL engine.

Stylus [48] exploits a strongly-typed key-value storage scheme called Microsoft Trinity Graph Engine [107][47]. It makes use of a data structure called **xUDT** for storing group of predicates identified by a unique id. An index which maps a given predicate to the unique id is maintained within Stylus. Another index is used to retrieve entities within triples. A random hashing is used to distribute RDF dataset over cluster of servers. In Stylus, the SPARQL query processing is reduced to sub-graph matching problem.

Koral [55] is a distributed system which offers the facility of alternate approaches for each component in distributed query processing. It is also based on the classic master-slave architecture. It implements various RDF graph partitioning strategies (hash-based and graph-based) for distributing data among computing nodes. SPARQL is used as a default query language in Koral. It is an extension of asynchronous execution mechanisms such as realised in TriAD. The query execution mechanism is independent from the underlying graph cover and can easily be replaced with another mechanism.

PRoST [24] stores RDF data both as VP and PT and hence take the advantages of both storage techniques with the cost of additional storage overhead. It does not maintain any additional indexes. SPARQL queries are translated into Join Tree format in which every node represents the VP table or PT's subquery's patterns. It makes use of the statistics-based query optimizer. The data is distributed using vertical partitioning.

WORQ [71] uses a query workload driven approach to partition the given RDF data among multiple data nodes which stores data in a Vertical partitioning manner. It uses Bloom Filters as indexes over the Vertically partitioned tables. The online reductions of RDF joins are performed by using bloom filters. In addition, caching of RDF joins reductions is performed to boost the query performance.

Anzograph[48] stores data in a graph form and entirely in memory. It can be installed on single or multiple nodes. It does not require an explcit index created by the user. SPARQL 1.1 queries are sent and received to different nodes over HTTP. Query processing follows a master-slave architecture. The query is issued at a master node which parse the query and assign to the query planner. The query planner decides the type of join or an aggregation needed for a query. Then the code generator unites the different steps into segments, after that all the segments are packaged for query into a stream. The master sends this stream to corresponding nodes in parallel. The results are then sent back to master and integrated into final query resultset.

Neptune[49] works as graph database service to work with highly connected datasets. Neptune stores RDF graph in

[47]https://github.com/Microsoft/GraphEngine.
[48]https://docs.cambridgesemantics.com/anzograph/userdoc/features.htm
[49]https://aws.amazon.com/neptune/

the form of a quad in Amazon S3. It implements three index permutations, SPOG, POGS and GPSO. In addition to three index permutations, each index permutation has 16 access patterns. Neptune stores all of its indexes in a Hash tables. Datasets in Neptune can be queried through SPARQL, Apache TinkerPop and Gremlin [50]. Neptune makes use of cardinality estimations and static analysis to rewrite queries, which will be executed through the pipeline of physical operators. In Neptune, there is no partitioning of graphs, instead it stores 15 replicas at its master node.

HF, VF, MF [84] (Adaptive Distributed RDF Graph Fragmentation and Allocation based on Query Workload) stores RDF data in three types of fragmentation: Horizontal Fragmentation (HF), Vertical Fragmentation (VF), and Mixed Fragmentation (MF). The data fragmentation is performed on the basis of frequently accessed patterns from the query workload. It implements no indexing mechanism. SPARQL query processing is based on the efficient fragmentation across different partitions. It implements a cost-based query planner.

DiStRDF [119] stores RDF data in an encoded form using unique integer identifiers. It employs a special purpose encoding scheme [117] for the encoding of RDF data stored in CSV or Parquet. DiStRDF can handle data stored both in TT or PT. It exploits range partitioning to distribute data. For indexing, DiStRDF makes use of the predicate pushdown mechanism offered by Spark with the combination of Parquet. This mechanism helps to select the required data by exploiting filters in the query. The query processing is based on Spark. Query processing exploits the encoding scheme to choose between different query execution plans. The physical implementation of logical plans is selected on the basis of static set of rules (rule-based optimization).

Leon [38] stores RDF data in a heuristics-based partitioning based on the characteristics set [77]. After encoding, Leon evenly distributes the triples across different partitions. It implements a bi-directional dictionary between id's in characteristics set and subjects. This dictionary is used as an index. Leon follows a master-slave architecture for query processing. By exploiting the characteristics set based partitioning, Leon minimizes the communication cost during query evaluation.

DISE [52] stores RDF data as 3D tensors, a multidimensional array of ordered columns . It performs the translation of SPARQL queries into Spark tensor operation through Spark-Scala compliant code.

10. SPARQL BENCHMARKS FOR RDF ENGINES

This discussion is adapted from [99] which analyzed different SPARQL benchmarks used to evaluate RDF engines. The discussion is divided into two sections: first, we explain the SPARQL benchmarks key design features, and then we present an analysis of the existing SPARQL benchmarks. The overall goal facilitates the development of benchmarks for high performance in the future, and help RDF engines developers/users to select the best SPARQL benchmark for the given scenario and RDF engine.

10.1 Benchmarks Design Features

[50]https://dbdb.io/db/neptune

SPARQL query benchmarks are comprised of three elements i.e., RDF datasets, SPARQL queries, and performance measures.

We first discussed the key features related to these elements that are vital to take in account in the development of RDF engine benchmarks. Then a high level analysis of the datasets and queries used in latest RDF engines benchmarks is presented.

Datasets. The datasets used in RDF engine benchmarks are of two types namely *synthetic* and *real-world* RDF datasets [97]. Real-world RDF datasets are very useful as they contain real world information [74]. On the other hand, synthetic datasets are helpful to test the scalability under varying size of datasets. Synthetic datasets of varying sizes are generated by different generators, which are tuned to reflect the features of real datasets [27]. In order to select a datasets for benchmarking RDF engines, two measures are proposed by [99]. These two measures are (1) Dataset Structuredness, (2) Relationship Speciality. The *structuredness* or *coherence* measures how well a instances of different dataset covers classes i.e., `rdf:type`. The value of structuredness of given dataset lies between 0 which stands for lowest possible structure and 1 points to a highest possible structured dataset. In RDF datasets, some resources are more distinguishable from others, and more attributes are more commonly associated with these resources. The count of different predicates related to each and every resource present in the dataset provides valuable insights about the graph structure of an RDF dataset and makes some resources distinguishable from others [87]. This type of *relationship speciality* is normal in real datasets. The relationship speciality of RDF datasets can be any positive natural number: the higher the number, the higher the relationship speciality. The value of both structuredness and relationship speciality of different datasets directly affect the result size, the number of intermediate results, and the selectivity values of the triple patterns in the SPARQL query [99]. The formal definitions of these datasets measures can be found in [99]. It is essential to mention that besides the dataset structuredness and relationship speciality, observations from the literature [94, 27] suggest that different features of RDF datasets should be considered during selection of datasets for SPARQL benchmarking. These feature for consideration are different number of triples, number of resources, number of properties, number of objects etc.

SPARQL Queries. Saleem et al. [99] suggests that a SPARQL querying benchmark should change queries in terms of different *query characteristics* i.e., the number of projection variables, number of triple patterns, result set sizes, query execution time, number of Basic Graph Patterns, number of join vertices, mean join vertex degree, mean triple pattern selectivities, join vertex types, and highly used SPARQL clauses etc. All of these characteristics impact on the runtime performance of RDF engines. Saleem et al. [99] proposed a composite measure containing these query features called the *Diversity Score* of the benchmark queries. The higher the diversity score, the more diverse the queries of the benchmark. The Diversity score is defined as follows.

Definition 9 (Queries Diversity Score). *Let μ_i presents the mean and σ_i is the standard deviation of a given distribution w.r.t. the i^{th} feature of the given distribution. Then the overall diversity score DS of the queries is calculated as the average coefficient of variation of all the features k in a* *query analyzed in the queries of benchmark B:*

$$DS = \frac{1}{k}\sum_{i=1}^{k}\frac{\sigma_i(B)}{\mu_i(B)}$$

Performance Mesures. Saleem et al. [99] divided the performance metrics for RDF engines comparison into four broader categories:

- **Query Processing Related**: These measures are related to query processing of the RDF engines, for which query execution time is the most important. However, since a benchmark usually contains many queries, reporting of the execution time for each and every query is not possible. For this problem, the combined results are presented in terms of measures like Query Mix per Hour (QMpH) and Queries per Second (QpS) [99]. Overheads like intermediate results, disk/memory swaps are important to measure during the query executions [105].
- **Data Storage Related:** This category includes the important performance measures like data loading time, acquired storage space, and size of index. [99].
- **Result Set Related:** A fair comparison between two RDF engines is only possible in case of production of same results. In this regard measures of correctness and completeness are considered [99].
- **Parallelism with/without Updates:** IGUANA, benchmarks execution framework [22] is used to measure triplestores capability of parallel query processing.

We now present a broader analysis of the different existing RDF engines benchmarks containing the above-mentioned features.

10.2 RDF Engines Benchmarks Analysis

Saleem et al. [99] considered the benchmarks according to the inclusion criteria. The criteria include things like the benchmark is used to evaluate query runtime performance evaluation of triplestores. The criteria also includes that both RDF data and SPARQL queries of the different benchmarks are publicly available or can be generated (3) the queries must not require reasoning to retrieve the complete results. RDF engines benchmarks are divided into two main categories of synthetic and real-data benchmarks.

10.2.0.1 Synthetic Benchmarks..

Synthetic benchmarks make use of the data (and/or query) generators to generate datasets and/or queries for benchmarking. Synthetic benchmarks are useful in testing the scalability of RDF engines with varying dataset sizes and querying workloads. However, such benchmarks can fail to show the features of real datasets or queries. The *Train Benchmark* (TrainBench) [115] covers different railway networks in terms of increasing sizes, and serialization of them in different formats including RDF. *The Waterloo SPARQL Diversity Test Suite* (WatDiv) [7] comes with a synthetic data generator which produces RDF data. It also produces an adjustable value of structuredness and a query generator. The queries are generated from different query templates. *SP2Bench* [105] covers the power-law distributions or Gaussian curves of the data present in the DBLP bibliographic database. *The Berlin SPARQL Benchmark* (BSBM) [13] makes use of different query templates to generate various number of SPARQL queries for the purpose of benchmarking related to multiple use cases such as explore, update, and business intelligence. *Bowlogna* [25] is used to model a real-world settings which are derived specifically from the Bologna process. Bologna mostly covers user needs in terms of analytic queries. The *LDBC Social Network Benchmark* (SNB) defines the *Interactive* workload (SNB-INT) to measure the evaluation of different graph patterns in a localized manner. This happens when the graph is being continuously updated [29]. LDBC also defines the *Business Intelligence* workload (SNB-BI), this workload cover a queries that mix aggregations with complex graph pattern matching with , covering a significant part of the graph [116], without any updates.

10.2.0.2 Real-Data Benchmarks..

Real-data benchmarks make use of real-world datasets and queries from real user query logs for benchmarking. Real-data benchmarks are useful in testing RDF engines more closely in real-world settings. However, such benchmarks may fail to test the scalability of RDF engines with varying dataset sizes and query workloads. *FEASIBLE* [97] is used as a cluster-based SPARQL benchmark generator and uses different SPARQL query logs. *DBpedia SPARQL Benchmark* (DBPSB) [74] is an another cluster based approach based on the generation of queries from logs of DBpedia, but it has a separate techniques used for clustering. Another dataset *FishMark* [11] dataset is obtained from FishBase[51] and provided in variant of both RDF and SQL. In FishBase, the log if web-based FishBase application provides the SPARQL queries. The performance of biological datasets are evaluated in *BioBench* [124]. It also evaluates the queries from five different real-world RDF datasets[52], i.e., Cell, Allie, PDBJ, DDBJ, and UniProt.

Basic Statistics. Table 11 shows high-level statistics of the selected datasets and queries of the benchmarks. In case of the synthetic benchmarks, data generators, the datasets chosen were also used in the original paper. In case of WatDiv, DBPSB, SNB, being a template-based query generators, one query per available template was selected. For FEASIBLE (a benchmark generation framework from query logs), 50 queries from DBpedia log was selected, for comparison with a WatDiv benchmark which comes with 20 basic query templates and 30 extensions for testing.[53]

Structuredness and Relationship Speciality. Figure 13a contains the structuredness values of the different benchmarks. Duan et al. [27] first proposed this measure and propose that the real-world datasets are low valued than synthetic benchmarks in structuredness. The results show that most of the synthetic benchmarks contain the structuredness feature. Even there are data generators, e.g WatDiv generator, allow the users to generate a benchmark dataset of according to their need of structuredness.

Figure 13b shows the relationship speciality values of different benchmarks. According to [87], it is evident that the values of relationship speciality shown by real world

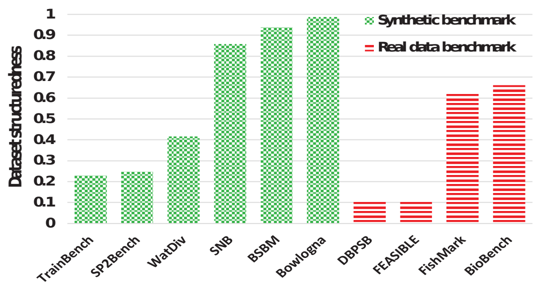
[51]FishBase: `http://fishbase.org/search.php`

[52]BioBench: `http://kiban.dbcls.jp/togordf/wiki/survey#data`

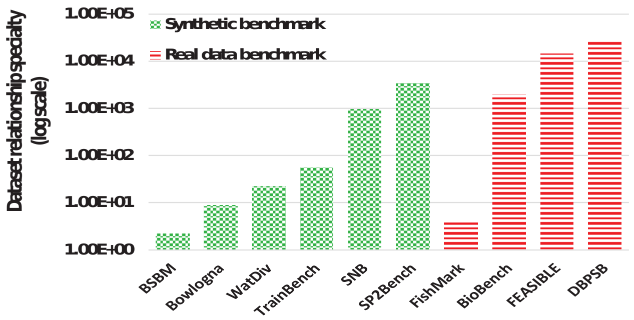
[53]WatDiv query templates: `http://dsg.uwaterloo.ca/watdiv/`

Table 11: Various high level statistics related to the data and queries used in the benchmarks. SNB represent SNB-BI and SNB-INT because of using the same dataset.

	Benchmark	Subjects	Predicates	Objects	Triples	Queries
Synthetic	Bowlogna [25]	2,151k	39	260k	12M	16
	TrainB. [115]	3,355k	16	3,357k	41M	11
	BSBM [13]	9,039k	40	14,966k	100M	20
	SP2Bench [105]	7,002k	5,718	19,347k	49M	14
	WatDiv [7]	5,212k	86	9,753k	108M	50
	SNB [29, 116]	7,193k	40	17,544k	46M	21
Real	FishMark [11]	395k	878	1,148k	10M	22
	BioBench [124]	278,007k	299	232,041k	1,451M	39
	FEASIBLE [97]	18,425k	39,672	65,184k	232M	50
	DBPSB [74]	18,425k	39,672	65,184k	232M	25



(a) Structuredness



(b) Relationship Specialty

Figure 13: Analysis of the different datasets used for RDF engines benchmarks.

datasets are higher than that of synthetic ones as shown in Figure 13b. This is also presented by average figures of (11098 vs 744). The relationship speciality values of real-world datasets are much higher than that synthetic ones like Bowlogna (8.7), BSBM (2.2), and WatDiv (22.0).

Diversity Score and Coverage. Figure 14 shows the overall diversity counts of the benchmarks. In real-data benchmarks, FEASIBLE generated benchmarks is the most diverse. While in synthetic benchmarks, WatDiv is the most diverse benchmark.

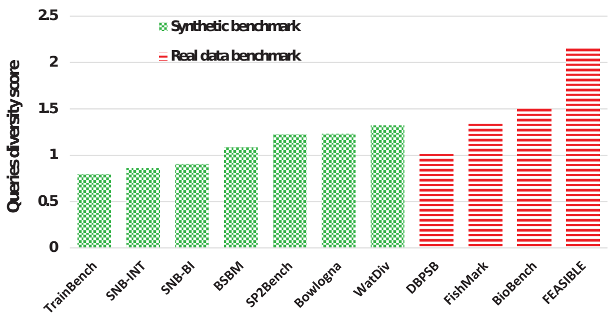


Figure 14: Benchmarks with diversity counts

Table 12 shows each benchmark with its coverage percentage of SPARQL clauses and join vertices [93]. The table also highlights the benchmarks with missing SPARQL constructs and types of overused join vertices. A lot of vital SPARQL constructs are missing from TrainBench and WatDiv queries. Every query of FishMark contains at least one Star join node.

Performance Measures Table 13 presents different benchmarks with performance measures to compare triplestores.

Which RDF engine is the fastest?. One of the critical questions is to know which RDF engine is the fastest in terms of query runtimes? According to our analysis, no other study compares both centralized and distributed RDF engines for their query runtime performances. The performance evaluations presented in the two most diverse SPARQL benchmarks, i.e. FEASIBLE[97] and WatDiv[6], showed that Virtuoso version 7.X is the fastest RDF engine. However, recently there are more RDF engines developed, and even existing engines are more improved in their new versions. As such, it would be interesting to perform a combined performance evaluation of both centralized and RDF engines.

11. RESEARCH PROBLEMS

This section illustrates the different research problems found during the study. Both centralized and distributed RDF engines lack the support for the update operation. Although in some setups it is implemented in a batch-wise manner, overall, it is immature. Indexing in RDF engines also has a lot of research potential. A number of indexes and the storage overhead caused by them require dedicated research efforts. To execute complex queries containing multiple triple patterns results in large intermediate results. These intermediate results cause network delay and I/O communications. This problem requires efficient storage and

Table 12: Percentages of different SPARQL clauses and types of join vertices in different benchmark. SPARQL clauses: `DIST[INCT]`, `FILT[ER]`, `REG[EX]`, `OPT[IONAL]`, `UN[ION]`, `LIM[IT]`, `ORD[ER BY]`. Join vertex types: Star, Path, Sink, Hyb[rid], N[o] J[oin]. Zeros and Hundreds represent the missing and overused features respectively.

		SPARQL Clauses							Types of Join vertices				
	Benchmark	`DIST`	`FILT`	`REG`	`OPT`	`UN`	`LIM`	`ORD`	Star	Path	Sink	Hyb.	N.J.
Synthetic	Bowlogna	6.2	37.5	6.2	0.0	0.0	6.2	6.2	93.7	37.5	62.5	25.0	6.2
	TrainB.	0.0	45.4	0.0	0.0	0.0	0.0	0.0	81.8	27.2	72.7	45.4	18.1
	BSBM	30.0	65.0	0.0	65.0	10.0	45.0	45.0	95.0	60.0	75.0	60.0	5.0
	SP2Bench	42.8	57.1	0.0	21.4	14.2	7.1	14.2	78.5	35.7	50.0	28.5	14.2
	Watdiv	0.0	0.0	0.0	0.0	0.0	0.0	0.0	28.0	64.0	26.0	20.0	0.0
	SNB-BI	0.0	61.9	4.7	52.3	14.2	80.9	100.0	90.4	38.1	80.9	52.3	0.0
	SNB-INT	0.0	47.3	0.0	31.5	15.7	63.15	78.9	94.7	42.1	94.7	84.2	0.0
Real	FEASIBLE	56.0	58.0	22.0	28.0	40.0	42.0	32.0	58.0	18.0	36.0	16.0	30.0
	Fishmark	0.0	0.0	0.0	9.0	0.0	0.0	0.0	100.0	81.8	9.0	72.7	0.0
	DBPSB	100.0	48.0	8.0	32.0	36.0	0.0	0.0	68.0	20.0	32.0	20.0	24.0
	BioBench	28.2	25.6	15.3	7.6	7.6	20.5	10.2	71.7	53.8	43.5	38.4	15.3

Table 13: Measures related to a query processing, data storage, result set, simultaneous multiple client requests, and dataset updates in different benchmarks. *QpS:* Queries per Second, *QMpH:* Queries Mix per Hour, *PO:* Processing Overhead, *LT:* Load Time, *SS:* Storage Space, *IS:* Index Sizes, *RCm:* Result Set Completeness, *RCr:* Result Set Correctness, *MC:* Multiple Clients, *DU:* Dataset Updates.

		Processing			Storage			Result Set		Additional	
	Benchmark	QpS	QMpH	PO	LT	SS	IS	RCm	RCr	MC	DU
Synthetic	Bowlogna	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗
	TrainBench	✗	✗	✗	✓	✗	✗	✓	✓	✗	✓
	BSBM	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓
	SP2Bench	✗	✗	✓	✓	✓	✗	✓	✓	✗	✗
	WatDiv	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	SNB-BI	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗
	SNB-INT	✓	✓	✗	✗	✗	✗	✓	✓	✗	✓
Real	FEASIBLE	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗
	Fishmark	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
	DBPSB	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
	BioBench	✗	✗	✗	✓	✓	✗	✓	✗	✓	✗

partitioning mechanism to distribute datasets in a way to avoid inter partition communication. It is also found during the study that upcoming distributed RDF engines use different prebuilt repositories with SQL interface to store RDF data. These systems perform translation of SPARQL or other domain query language into SQL for applying queries on the repositories. This translation is the main component in their operation, but SPARQL query optimization was not their core aspect. The translation process does not take into account the different features related to the distribution of datasets among a cluster of machines. The translation process also does not take into account the other Semantic Web concepts of Ontologies and Inference etc.

12. CONCLUSION

This paper reviews centralized and distributed RDF engines. We review both categories in terms of their storage, indexing, language, and query execution. Due to an ever increasing size of RDF based data, centralized RDF engines are becoming ineffective for interactive query processing. For the problems of effective query processing, distributed RDF engines are in use. Rather than relying on the specialized storage mechanism, some distributed RDF engines rely on existing single-node systems to take advantage of the underlying infrastructure. SQL is a choice for query language in most of the repositories. Rather than making capable SQL for RDF data, RDF engines translate SPARQL into other languages like SQL, PigLatin etc. This translation is an integral step in distributed setups.

13. ADDITIONAL AUTHORS

14. REFERENCES

[1] Chapter five - storage and indexing of rdf data. In O. CurÃ© and G. Blin, editors, *RDF Database Systems*, pages 105 – 144. Morgan Kaufmann, Boston, 2015.

[2] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *33rd International Conference on Very Large Data Bases, VLDB 2007 - Conference Proceedings*, VLDB '07, pages 411–422. VLDB Endowment, 2007.

[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] 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. Springer, 2018.

[5] K. Alaoui. A categorization of rdf triplestores. In *Proceedings of the 4th International Conference on Smart City Applications*, SCA '19, New York, NY, USA, 2019. Association for Computing Machinery.

[6] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of rdf data management systems. In *International Semantic Web Conference*, pages 197–212. Springer, 2014.

[7] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212. 2014.

[8] G. Aluç, M. T. Özsu, K. Daudjee, and O. Hartig. chameleon-db: a workload-aware robust rdf data management system. 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*, CIKM '12, page 2749–2751, New York, NY, USA, 2012. Association for Computing Machinery.

[10] 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 (SSWS2009)*, page 33, 2009.

[11] S. Bail, S. Alkiviadous, B. Parsia, D. Workman, M. van Harmelen, R. S. Gonçalves, and C. Garilao. FishMark: A linked data application benchmark. In *Proceedings of the Joint Workshop on Scalable and High-Performance Semantic Web Systems*, pages 1–15, 2012.

[12] D. Beckett. The design and implementation of the redland RDF application framework. In *Proceedings of the 10th International Conference on World Wide Web, WWW 2001*, WWW '01, pages 449–456, New York, NY, USA, 2001. ACM.

[13] C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[14] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 121–132, New York, NY, USA, 2013. Association for Computing Machinery.

[15] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. Dogma: A disk-oriented graph matching algorithm for rdf databases. In A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, and K. Thirunarayan, editors, *The Semantic Web - ISWC 2009*, pages 97–113, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[16] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In I. Horrocks and J. Hendler, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2342 LNCS, pages 54–68, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[17] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning, 2013.

[18] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao. Ontop: Answering sparql queries over relational databases. *Semantic Web*, 8(3):471–487, 2017.

[19] X. Chen, H. Chen, N. Zhang, and S. Zhang. Sparkrdf: Elastic discreted rdf graph processing engine with distributed memory. In *Proceedings of the 2014 International Conference on Posters and Demonstrations Track - Volume 1272*, ISWC-PD'14, page 261–264, Aachen, DEU, 2014. CEUR-WS.org.

[20] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, page 1216–1227. VLDB Endowment, 2005.

[21] F. Conrads, J. Lehmann, M. Saleem, M. Morsey, and A.-C. N. Ngomo. I guana: a generic framework for benchmarking the read-write performance of triple stores. In *International Semantic Web Conference*, pages 48–65. Springer, 2017.

[22] F. Conrads, J. Lehmann, M. Saleem, M. Morsey, and A. N. Ngomo. IGUANA: A generic framework for benchmarking the read-write performance of triple stores. In *ISWC*, pages 48–65. Springer, 2017.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[24] M. Cossu, M. Färber, and G. Lausen. Prost: Distributed execution of SpaRQL queries using mixed partitioning strategies. In *Advances in Database Technology - EDBT*, volume 2018-March, pages 469–472, 2018.

[25] G. Demartini, I. Enchev, M. Wylot, J. Gapany, and P. Cudré-Mauroux. BowlognaBench - benchmarking RDF analytics. In *Data-Driven Process Discovery and Analysis SIMPDA*, pages 82–102. Springer, 2011.

[26] U. Deppisch. S-tree: A dynamic balanced signature index for office retrieval. In *Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '86, page 77–87, New York, NY, USA, 1986. Association for Computing Machinery.

[27] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: A comparison of RDF benchmarks and real RDF datasets. In *SIGMOD*, pages 145–156. ACM, 2011.

[28] 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.

[29] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi,

A. Gubichev, A. Prat-Pérez, M. Pham, and P. A. Boncz. The LDBC Social Network Benchmark: Interactive workload. In *SIGMOD*, pages 619–630. ACM, 2015.

[30] O. Erling and I. Mikhailov. *Virtuoso: RDF Support in a Native RDBMS*, pages 501–519. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[31] D. C. Faye, O. Curé, and G. Blin. A survey of RDF storage approaches. In *Revue Africaine de la Recherche en Informatique et Math{é}matiques Appliqu{é}es*, volume 15, page pp. 25, 2012.

[32] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary rdf representation for publication and exchange (hdt). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19:22–41, 2013.

[33] L. Galárraga, K. Hose, and R. Schenkel. Partout: A distributed engine for efficient RDF processing. In *WWW 2014 Companion - Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, pages 267–268, New York, NY, USA, 2014. ACM.

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

[35] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[36] D. Graux, L. Jachiet, P. Genevès, and N. Layaïda. SPARQLGX: Efficient distributed evaluation of SPARQL with apache spark. In P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, and Y. Gil, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9982 LNCS, pages 80–87, Cham, 2016. Springer International Publishing.

[37] J. Groff and P. Weinberg. *SQL The Complete Reference, 3rd Edition*. McGraw-Hill, Inc., USA, 3 edition, 2009.

[38] X. Guo, H. Gao, and Z. Zou. Leon: A distributed rdf engine for multi-query processing. In G. Li, J. Yang, J. Gama, J. Natwichai, and Y. Tong, editors, *Database Systems for Advanced Applications*, pages 742–759, Cham, 2019. Springer International Publishing.

[39] 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 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 289–300, New York, NY, USA, 2014. ACM.

[40] 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, feb 2015.

[41] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB Journal*, 25(3):355–380, jun 2016.

[42] S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, pages 1–20, 2003.

[43] S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In *CEUR Workshop Proceedings*, volume 517, pages 94–109, 2009.

[44] A. Harth. Cumulusrdf: Linked data management on nested key-value stores. 2011.

[45] A. Harth and S. Decker. Optimized index structures for querying RDF from the Web. In *Proceedings - Third Latin American Web Congress, LA-WEB 2005*, volume 2005 of *LA-WEB '05*, pages 71–80, Washington, DC, USA, 2005. IEEE Computer Society.

[46] 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 The Semantic Web and 2nd Asian Conference on Asian Semantic Web Conference*, ISWC'07/ASWC'07, page 211–224, Berlin, Heidelberg, 2007. Springer-Verlag.

[47] O. Hartig and M. T. Özsu. Linked Data query processing. In *Proceedings - International Conference on Data Engineering*, pages 1286–1289, mar 2014.

[48] L. He, B. Shao, Y. Li, H. Xia, Y. Xiao, E. Chen, and L. J. Chen. Stylus: A strongly-typed store for serving massive rdf data. *Proc. VLDB Endow.*, 11(2):203–216, Oct. 2017.

[49] K. Hose and R. Schenkel. WARP: Workload-aware replication and partitioning for RDF. In *Proceedings - International Conference on Data Engineering*, pages 1–6, apr 2013.

[50] 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.

[51] V. Ingalalli, D. Ienco, and P. Poncelet. *Querying RDF Data: a Multigraph-based Approach*, chapter 5, pages 135–165. John Wiley Sons, Ltd, 2018.

[52] H. Jabeen, E. Haziiev, G. Sejdiu, and J. Lehmann. Dise: A distributed in-memory sparql processing engine over tensor data. In *2020 IEEE 14th International Conference on Semantic Computing (ICSC)*, pages 400–407, 2020.

[53] M. Janik and K. Kochut. Brahms: a workbench rdf store and high performance memory system for semantic association discovery. In *International Semantic Web Conference*, pages 431–445. Springer, 2005.

[54] D. Janke and S. Staab. Storing and querying semantic data in the cloud. In *Reasoning Web International Summer School*, pages 173–222. Springer, 2018.

[55] D. Janke, S. Staab, and M. Thimm. Koral: A glass box profiling system for individual components of distributed rdf stores. In *BLINK/NLIWoD3@ISWC*, 2017.

[56] D. Janke, S. Staab, and M. Thimm. On data placement strategies in distributed rdf stores. In

Proceedings of The International Workshop on Semantic Big Data, SBD '17, New York, NY, USA, 2017. Association for Computing Machinery.

[57] D. D. Janke. *Study on Data Placement Strategies in Distributed RDF Stores*, volume 46. IOS Press, 2020.

[58] Z. Kaoudi and I. Manolescu. Rdf in the clouds: a survey. *The VLDB Journal*, 24(1):67–91, 2015.

[59] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20(1):359–392, dec 1998.

[60] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[61] A. Katib, V. Slavov, and P. Rao. Riq: Fast processing of sparql queries on rdf quadruples. *Journal of Web Semantics*, 37:90–111, 2016.

[62] 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 and Demonstrations Track - Volume 914*, ISWC-PD'12, page 85–88, Aachen, DEU, 2012. CEUR-WS.org.

[63] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi. Taming subgraph isomorphism for rdf query processing. *Proceedings of the VLDB Endowment*, 8(11), 2015.

[64] D. Kolas, I. Emmons, and M. Dean. Efficient linked-list RDF indexing in Parliament. *CEUR Workshop Proceedings*, 517:17–32, 2009.

[65] M. Koubarakis and K. Kyzirakos. Modeling and querying metadata in the semantic sensor web: The model strdf and the query language stsparql. In *Proceedings of the 7th International Conference on The Semantic Web: Research and Applications - Volume Part I*, ESWC'10, page 425–439, Berlin, Heidelberg, 2010. Springer-Verlag.

[66] K. Kyzirakos, M. Karpathiotakis, and M. Koubarakis. Strabon: A semantic geospatial dbms. In P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, editors, *The Semantic Web – ISWC 2012*, pages 295–311, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

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

[68] J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin, N. Chakraborty, M. Saleem, A.-C. Ngonga Ngomo, and H. Jabeen. Distributed semantic analytics using the sansa stack. In C. d'Amato, M. Fernandez, V. Tamma, F. Lecue, P. Cudré-Mauroux, J. Sequeda, C. Lange, and J. Heflin, editors, *The Semantic Web – ISWC 2017*, pages 147–155, Cham, 2017. Springer International Publishing.

[69] 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, 2004.

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

[71] A. Madkour, A. M. Aly, and W. G. Aref. Worq: Workload-driven rdf query processing. In D. Vrandečić, K. Bontcheva, M. C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L.-A. Kaffee, and E. Simperl, editors, *The Semantic Web – ISWC 2018*, pages 583–599, Cham, 2018. Springer International Publishing.

[72] B. McBride. Jena: A semantic web toolkit. *IEEE Internet Computing*, 6(6):55–58, nov 2002.

[73] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[74] M. Morsey, J. Lehmann, S. Auer, and A. N. Ngomo. DBpedia SPARQL benchmark - performance assessment with real queries on real data. In *ISWC*, pages 454–469. Springer, 2011.

[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 and Demonstrations Track) - Volume 1035*, ISWC-PD '13, page 261–264, Aachen, DEU, 2013. CEUR-WS.org.

[76] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. Rdfox: A highly-scalable rdf store. In M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, and S. Staab, editors, *The Semantic Web - ISWC 2015*, pages 3–20, Cham, 2015. Springer International Publishing.

[77] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*, pages 984–994, 2011.

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

[79] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

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

[81] 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, oct 2018.

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

[83] 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*, WWW

'12 Companion, page 397–400, New York, NY, USA, 2012. Association for Computing Machinery.

[84] P. Peng, L. Zou, L. Chen, and D. Zhao. Adaptive distributed rdf graph fragmentation and allocation based on query workload. *IEEE Transactions on Knowledge and Data Engineering*, 31(4):670–685, 2019.

[85] P. Peng, L. Zou, M. T. Özsu, L. Chen, and D. Zhao. Processing sparql queries over distributed rdf graphs. *The VLDB Journal*, 25(2):243–268, Apr. 2016.

[86] R. Punnoose, A. Crainiceanu, and D. Rapp. Rya: A scalable rdf triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, Cloud-I '12, New York, NY, USA, 2012. Association for Computing Machinery.

[87] S. Qiao and Z. M. Özsoyoglu. RBench: Application-specific RDF benchmarking. In *SIGMOD*, pages 1825–1838. ACM, 2015.

[88] M. O. RABIN. Fingerprinting by random polynomials. *Technical Report*, 1981.

[89] P. Ravindra, H. Kim, and K. Anyanwu. An intermediate algebra for optimizing rdf graph pattern matching on mapreduce. In G. Antoniou, M. Grobelnik, E. Simperl, B. Parsia, D. Plexousakis, P. De Leenheer, and J. Pan, editors, *The Semantic Web: Research and Applications*, pages 46–61, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[90] 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*, PSI EtA '10, New York, NY, USA, 2010. Association for Computing Machinery.

[91] S. Sakr and G. Al-Naymat. Relational processing of rdf queries: a survey. *ACM SIGMOD Record*, 38(4):23–28, 2010.

[92] S. Sakr, M. Wylot, R. Mutharaju, D. Le Phuoc, and I. Fundulaki. *Linked Data: Storing, Querying, and Reasoning*. Springer, 2018.

[93] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A. N. Ngomo. LSQ: the linked SPARQL queries dataset. In *ISWC*, pages 261–269. Springer, 2015.

[94] M. Saleem, A. Hasnain, and A. N. Ngomo. LargeRDFBench: A billion triples benchmark for SPARQL endpoint federation. *J. Web Sem.*, 48:85–125, 2018.

[95] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. Ngonga Ngomo. A fine-grained evaluation of sparql endpoint federation systems. *Semantic Web*, 7(5):493–518, 2016.

[96] M. Saleem, Q. Mehmood, and A.-C. N. Ngomo. Feasible: A feature-based sparql benchmark generation framework. In *International Semantic Web Conference*, pages 52–69. Springer, 2015.

[97] M. Saleem, Q. Mehmood, and A. N. Ngomo. FEASIBLE: a feature-based SPARQL benchmark generation framework. In *ISWC*, pages 52–69. Springer, 2015.

[98] M. Saleem, A. Potocki, T. Soru, O. Hartig, and A. N. Ngomo. CostFed: Cost-based query optimization for SPARQL endpoint federation. In *SEMANTICS*, volume 137 of *Procedia Computer Science*, pages 163–174. Elsevier, 2018.

[99] M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood, and A.-C. Ngonga Ngomo. How representative is a sparql benchmark? an analysis of rdf triplestore benchmarks. In *The World Wide Web Conference*, WWW '19, page 1623–1633, New York, NY, USA, 2019. Association for Computing Machinery.

[100] L. H. Z. Santana and R. d. S. Mello. An analysis of mapping strategies for storing rdf data into nosql databases. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 386–392, New York, NY, USA, 2020. Association for Computing Machinery.

[101] A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen. In F. Wang, G. Luo, C. Weng, A. Khan, P. Mitra, and C. Yu, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9579, pages 155–168, Cham, 2016. Springer International Publishing.

[102] 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, SWIM 2011*, 2011.

[103] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen. Sempala: Interactive SPARQL query processing on Hadoop. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8796, pages 164–179, 2014.

[104] 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.

[105] M. Schmidt et al. SP2Bench: A SPARQL performance benchmark. In *Semantic Web Information Management - A Model-Based Perspective*, pages 371–393. 2009.

[106] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516, 2013.

[107] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 505–516, New York, NY, USA, 2013. Association for Computing Machinery.

[108] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 317–332, USA, 2016. USENIX Association.

[109] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[110] C. Stadler, G. Sejdiu, D. Graux, and J. Lehmann. Sparklify: A scalable software component for efficient evaluation of sparql queries over distributed rdf datasets. In C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois, and F. Gandon, editors, *The Semantic Web – ISWC 2019*, pages 293–308, Cham, 2019. Springer International Publishing.

[111] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, page 553–564. VLDB Endowment, 2005.

[112] P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: Graph algorithms for the (semantic) web. In P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, I. Horrocks, and B. Glimm, editors, *The Semantic Web – ISWC 2010*, pages 764–780, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[113] P. Stutz, M. Verman, L. Fischer, and A. Bernstein. Triplerush: A fast and scalable triple store. In *Proceedings of the 9th International Conference on Scalable Semantic Web Knowledge Base Systems - Volume 1046*, SSWS'13, page 50–65, Aachen, DEU, 2013. CEUR-WS.org.

[114] M. Svoboda and I. Mlŷnková. Linked data indexing methods: A survey. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 474–483. Springer, 2011.

[115] G. Szárnyas, B. Izsó, I. Ráth, and D. Varró. The Train Benchmark: Cross-technology performance evaluation of continuous model queries. *Softw. Syst. Model.*, 17(4):1365–1393, 2018.

[116] G. Szárnyas, A. Prat-Pérez, A. Averbuch, J. Marton, M. Paradies, M. Kaufmann, O. Erling, P. A. Boncz, V. Haprian, and J. B. Antal. An early look at the LDBC Social Network Benchmark's Business Intelligence workload. In *GRADES-NDA at SIGMOD*, pages 9:1–9:11. ACM, 2018.

[117] A. Vlachou, C. Doulkeridis, A. Glenis, G. M. Santipantakis, and G. A. Vouros. Efficient spatio-temporal rdf query processing in large dynamic knowledge bases. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 439–447, New York, NY, USA, 2019. Association for Computing Machinery.

[118] 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.

[119] R. T. Whitman, B. G. Marsh, M. B. Park, and E. G. Hoel. Distributed spatial and spatio-temporal join on apache spark. *ACM Trans. Spatial Algorithms Syst.*, 5(1), June 2019.

[120] K. Wilkinson. Jena property table implementation, 2006.

[121] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in jena2. In *Proceedings of the 1st International Conference on Semantic Web and Databases, SWDB 2003*, SWDB'03, pages 120–139, Aachen, Germany, Germany, 2003. CEUR-WS.org.

[122] D. Wood, P. Gearon, and T. Adams. Kowari: A platform for semantic web storage and analysis. In *XTech 2005 Conference*, pages 5–402, 2005.

[123] B. Wu, Y. Zhou, P. Yuan, H. Jin, and L. Liu. Semstore: A semantic-preserving distributed rdf triple store. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, page 509–518, New York, NY, USA, 2014. Association for Computing Machinery.

[124] H. Wu et al. BioBenchmark Toyama 2012: An evaluation of the performance of triple stores on biological data. *J. Biomedical Semantics*, 5:32, 2014.

[125] 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.

[126] M. Wylot, M. Hauswirth, P. Cudré-Mauroux, and S. Sakr. Rdf data storage and query processing schemes: A survey. *ACM Comput. Surv.*, 51(4):84:1–84:36, Sept. 2018.

[127] M. Wylot, J. Pont, M. Wisniewski, and P. Cudré-Mauroux. Diplodocus[rdf]: Short and long-tail rdf analytics for massive webs of data. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, page 778–793, Berlin, Heidelberg, 2011. Springer-Verlag.

[128] M. Q. Yasin, X. Zhang, R. Haq, Z. Feng, and S. Yitagesu. A comprehensive study for essentiality of graph based distributed sparql query processing. In *International Conference on Database Systems for Advanced Applications*, pages 156–170. Springer, 2018.

[129] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: A fast and compact system for large scale rdf data. *Proc. VLDB Endow.*, 6(7):517–528, May 2013.

[130] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[131] 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 of *PVLDB'13*, pages 265–276. VLDB Endowment, 2013.

[132] X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *Proceedings - International Conference on Data Engineering*, pages 565–576, apr 2013.

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