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

A Preprint

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

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

May 30, 2020

Abstract

The recent advancements of the Semantic Web and Linked Data have changed the working of the traditional web. There is a huge 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 different mechanisms to implement key 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 key for fast data lookup. The type of the underlying querying language (e.g., SPARQL or SQL) used for query execution is a key 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.

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 which store their data in 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 are inefficient to meet the growing demand of storage, computing capacity and processing of complex SPARQL queries [1, 2, 3, 4]. To tackle this issue different kinds of distributed RDF engines were proposed [5, 6, 7, 8, 3, 9, 10]. These distributed systems run on a set of cluster hardware containing a different number of machines with dedicated memory and storage.

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 [2]. Recent evaluation [11] 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.

- **Querying Language.** Various RDF engines store data in different formats, thus support various querying languages such as SQL [12], PigLatin [13] etc. Since SPARQL is used as the standard query language, 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.

There are various studies that categorize, compare, and evaluate different RDF engines. For example, the query runtime evaluation of different RDF engines are shown in [2, 4, 14, 15, 16]. Studies like [17, 18] are focused towards the data storage mechanisms in RDF engines. Svoboda et al. [19] classify various indexing approaches used for linked data. The usage of relational data models for RDF data is presented in [20]. A survey of the RDF on the cloud is presented in [21]. A high level illustration of the different centralized and distributed RDF engines and linked data query techniques are presented in [22]. Finally, empirical performance evaluation and a broader overview of the distributed RDF engines are presented in [4]. 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 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 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

---

[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/.

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 the brief explanation of RDF and SPARQL. Main purpose of explanation is to establish the basic understanding of the terminologies used in paper. For complete details, readers are encouraged to look at original W3C sources of RDF[6] and SPARQL[7]. This discussion is adapted from [22, 23, 3, 24].

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.

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* $D = \{< s_1, p_1, o_1 >, \ldots, < s_n, p_n, o_n >\}$.

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

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"

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.

---

[6]RDF Primer: `http://www.w3.org/TR/rdf-primer/`.

[7]SPARQL Specification: `https://www.w3.org/TR/sparql11-query/`

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 [22].

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 vertexes 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 $f_V : V \rightarrow L_V$ 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 $f_E : E \rightarrow L_E$ 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.

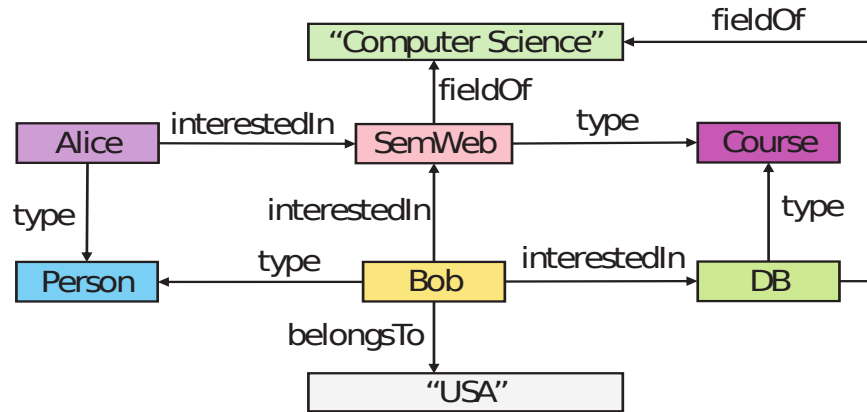



Fig. 1. Pictorial representation of graph

2.2 SPARQL

SPARQL [25] 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) [23], a generalization of a directed graph in which a hyperedge can join any number of vertices. In this representation, every hyperedge captures a triple pattern. The subject of the triple becomes the source vertex of a hyperedge, and the predicate and object of the triple pattern become the target vertices. 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. As a result, we can capture joins that involve predicates of triple patterns. Formally, the hypergraph representation is defined as follows:

---

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

Definition 4 (Directed hypergraph of a BGP) *The hypergraph representation of a BGP B is 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 $T = (p, o)$.*

The representation of a complete SPARQL query as a DH is the union of the representations of the query's BGPs.

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

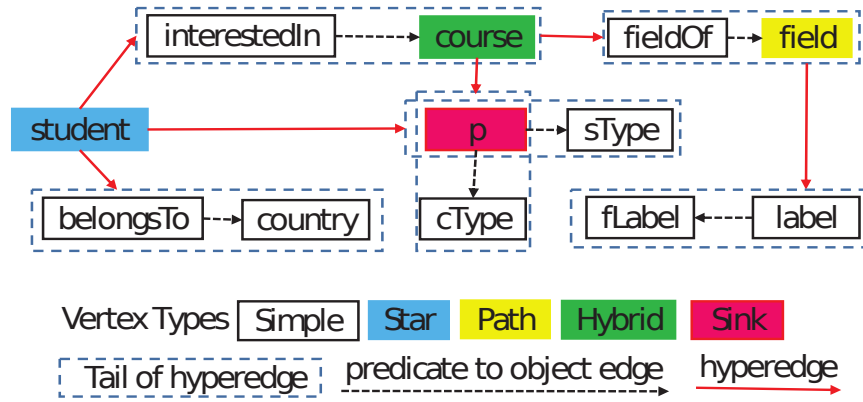



Figure 2: Directed hypergraph representation of a SPARQL query. Prefixes are ignored for simplicity.

Based on the DH representation of SPARQL queries, we can define the following features of SPARQL queries:

Definition 5 (Join Vertex) *For every vertex $v \in V$ in such a hypergraph we write $E_{in}(v)$ and $E_{out}(v)$ to denote the set of incoming and outgoing edges, respectively; i.e., $E_{in}(v) = \{(S, T) \in E \mid v \in T\}$ and $E_{out}(v) = \{(S, T) \in E \mid v \in S\}$. If $|E_{in}(v)| + |E_{out}(v)| > 1$, 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. [20] 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 [22]. A survey of the storage of RDF data in relational and NoSQL database stores is presented in [18]. Pan et al. [2] 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. [26]. They also discussed various SPARQL benchmarks.

Svoboda et al. [19] 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. [17] 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 permanently store RDF data. 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 [1]. This study focuses on data storage techniques, indexing strategies, and query execution mechanisms.

Kaoudi et al. [21] 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. [27] presented another survey on the storage and query processing techniques used in the RDF engines on the cloud. Janke et al. [28, 29] 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 [4]. 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. [30] discussed the limitations and discrepancies in distributed RDF engines. In particular, they discussed the SPARQL 1.1 support in these engines. Authors in [31] 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 [32], 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.

Study	Year	Central Focus
Janke et al. [29]	2020	**Partitioning**, **indexing**, and **query evaluation** in **distributed** RDF engines
Santana et al. [32]	2020	Discussion of different **Mapping** strategies for storage of RDF based data into the NoSQL databases
Alaoui [31]	2019	General categorization of different RDF engines
Janke et al. [28]	2018	**Partitioning**, **indexing**, and **query evaluation** in **distributed** and **cloud** RDF engines
Sakr et. al [26]	2018	**Storage** and **query evaluation** in **centralized** and **distributed** RDF engines
Wylot et. al [1]	2018	**Storage** and **indexing** in **centralized** and **distributed** RDF engines
Pan et al. [2]	2018	**Storage** techniques in **centralized** and **distributed** RDF engines
Yasin et al. [30]	2018	Suitability and **query evaluation** of **distributed** RDF engines
Elzein et al. [27]	2018	**Storage** and **query evaluation** in RDF engines on **cloud**
Abdelaziz et al. [4]	2017	**Performance** evaluation of **distributed** RDF engines
Ma et al. [18]	2016	**Storage** techniques in **centralized** RDF engines
Ozsu [22]	2016	General overview of **centralized** and **distributed** RDF engines
Kaoudi et al. [21]	2015	RDF engines on **cloud**
Faye et al. [17]	2012	**Storage** and **Indexing** techniques in **centralized** RDF engines
Svoboda et al. [19]	2011	**Indexing** in **centralized**, **distributed**, and global RDF engines
Sakr et al. [20]	2010	**Storage** and **query evaluation** in SPARQL transnational RDF engines

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

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 the state-of-the-art SPARQL benchmarks designed for the performance evaluation of RDF engines. We show the pro and cons of these benchmarks, which will help the user to choose the best representative benchmark for the given use-case.

---

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

4   Storage

Data storage is an integral component of any Database Management System (DBMS). Efficient data storage is key 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.

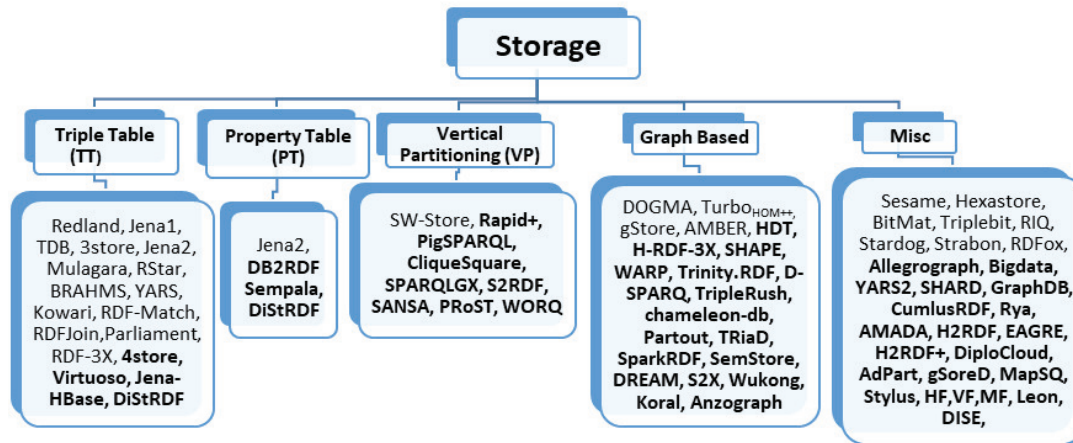



Figure 3: Pictorial representation of Storage in Centralized and Distributed RDF Engines (*RDF Engines in bold text are Distributed engines)

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.

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"

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

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 [33]) only make use of the dictionary tables to store strings whose lengths exceed a threshold. On the one hand, this allows filters operation to be performed directly on the TT. However, on the other hand, it also results in higher storage consumption due to string values are stored multiple times.

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

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"

(a) Integer Triple Table using dictionary   (b) Dictionary

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

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 engines[11] such as Virtuoso [34] and 4store [35]. 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.

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

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

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

---

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

[11]Further details given in section section 9.

[12]Specifying RDF datasets: `https://www.w3.org/TR/sparql11-query/#specifyingDataset`

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 see that TT often comes with various indexes, mostly six triple permutations $\mathbf{P}(<s,p,o>) = \mathbf{spo, sop, pso, pos, ops, osp}$.

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 [36] suggest that many real-world RDF datasets contain missing information for different resources.

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

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

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

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

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.

Subject	type	belongsTo
Bob	Person	"USA"
Alice	Person	null

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

Subject	interestedIn
Bob	SemWeb
Bob	DB
Alice	SemWeb

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

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

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

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.

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 [37], show that 33% of the real-world queries contain *star* join, 8.79% 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 [33], 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 [38]. 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.

Subject	Object
Bob	Person
Alice	Person
SemWeb	Course
DB	Course

(a) Predicate `type`

Subject	Object
Bob	SemWeb
Bob	DB
Alice	SemWeb

(b) Predicate `interestedIn`

Subject	Object
Bob	"USA"

(c) Predicate `belongsTo`

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

(d) Predicate `fieldOf`

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

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, [39] store RDF graphs as directed signature graphs stored as a disk-based adjacency list table, [40] store as balanced binary tree, and [41] 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.

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.

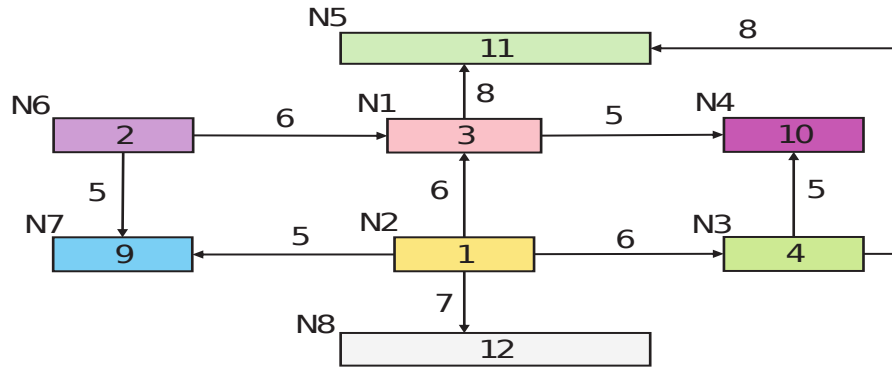



Fig. 4. Signature graph representation using dictionary encoding shown in Table 4b

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 [42], HBase tables based[13] [43], API based like Sesame SAIL Storage And Inference Layer [44], on disk or in memory-based [45], and Bit Matrix-based [46]. 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.

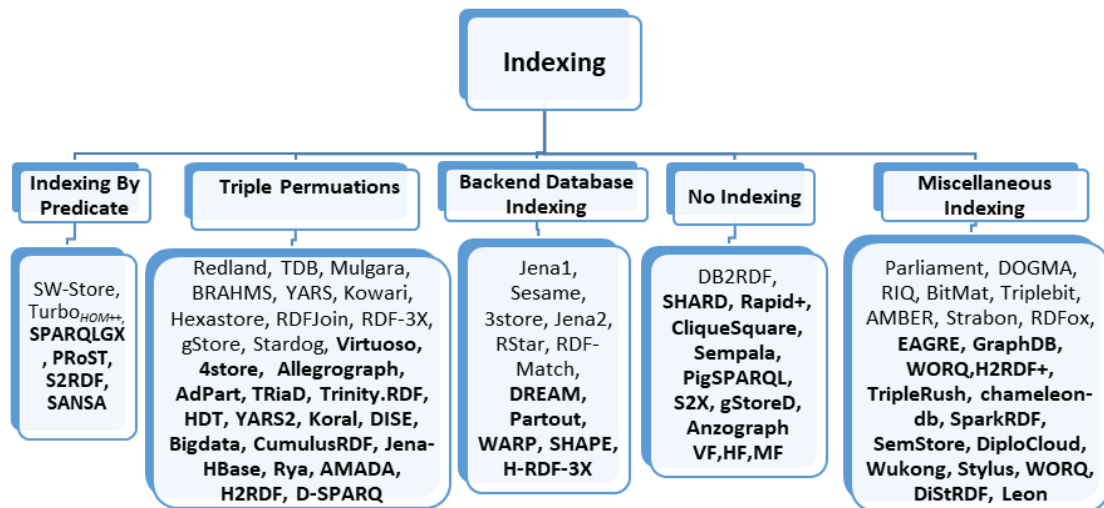



Figure 5: Pictorial representation of Indexing schemes in Centralized and Distributed RDF Engines (*RDF Engines in bold text are Distributed engines)

- **Indexing By Predicate**
  This type of indexing is found in RDF engines where the data is stored as Vertically Partitioned (VP) tables. Each of the VP table is naturally indexed by predicate. Predicate-based indexing is particularly helpful for

---

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

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. $<:s, ?p, ?o>, <?s, ?p, :o>, and <:s, ?p, :o>$.

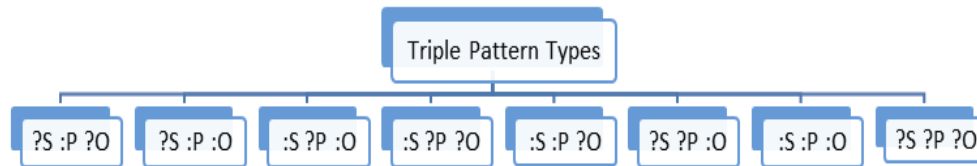



Fig. 6. Different types of SPARQL triple patterns based on (un)bound subject S , predicate P and object O. The ":" refers to bound and "?" refers to unbound subject, predicate or object of a triple pattern.

- **Triple-Permutation Indexing.** As mentioned before, an RDF triple comprises Subject, Predicate, and Object. Figure 7 shows the permutations of an RDF triple $\langle s,p,o \rangle$. The RDF engines in this category maintain 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 $V_P = \{p_1^i, \ldots, p_n^i\}$ in maintained; and for each element $p_j^i \in V_P$, a separate list $L_O = \{o_1^{i,j}, \ldots, o_k^{i,j}\}$ 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.

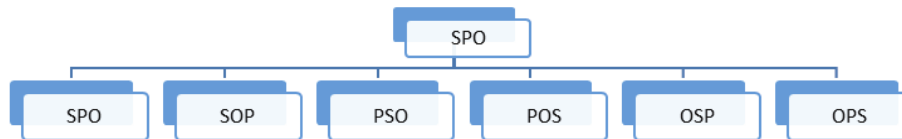



Fig. 7. Common Triple-Permutation indexes in RDF-3X and Hexastore.

- **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.

---

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

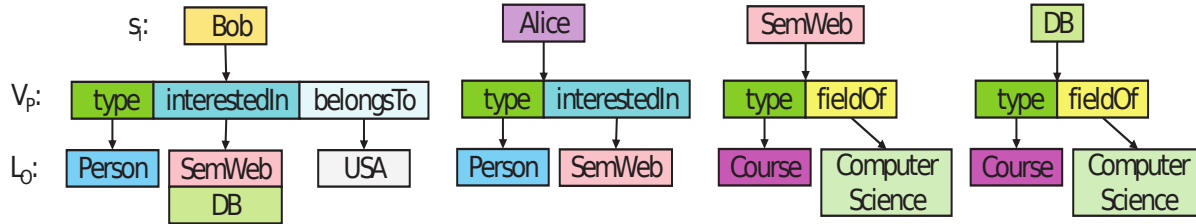



Figure 8: Simple SPO index for the sample RDF dataset given in Table 1.

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 maintain any indexing scheme at all. This could be due to the storage solution they used can be 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 [47], Space-filling [42], array indexing [48] and Hbase tables[15] [43] 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 [49], Ontop [50] etc.

3. **Other Languages**: Some RDF engines also use their own query language other than SPARQL i.e., RDF Data Query Language (RDQL) [51], 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 [45].

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

---

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

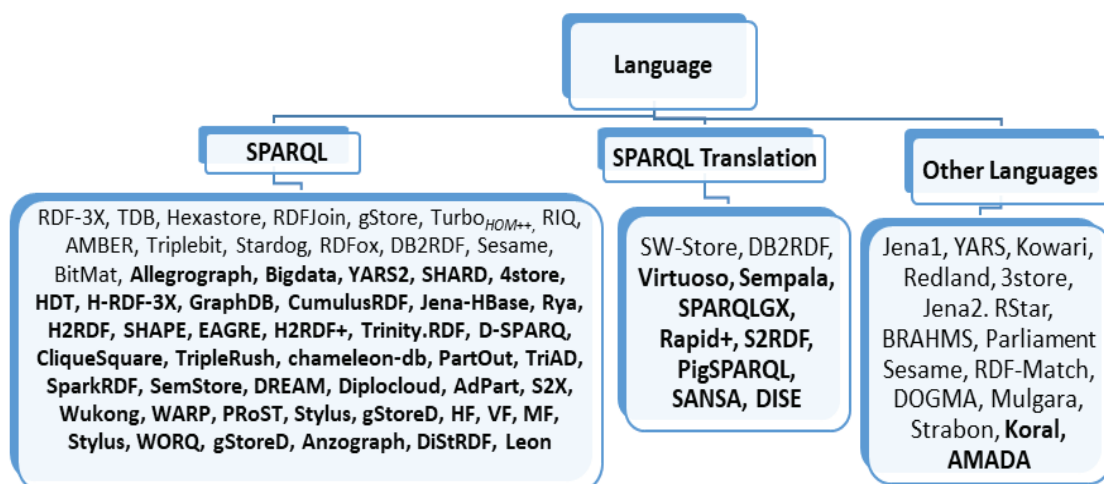



Figure 9: Pictorial representation of Query languages in Centralized and Distributed RDF Engines (*RDF Engines in bold text are Distributed engines)

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.

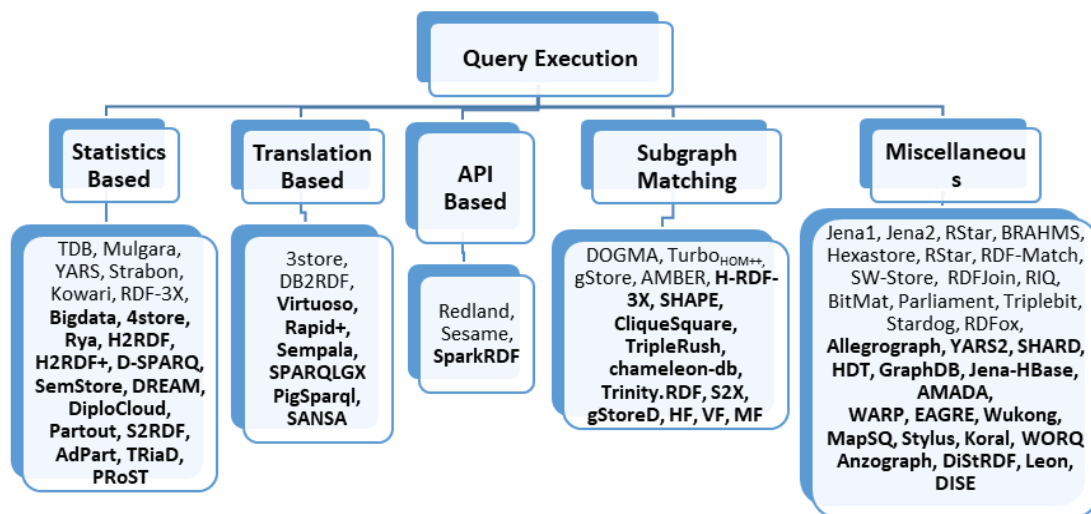



Fig. 10. Types of Query Execution in Centralized and Distributed RDF Engines(*Bold represents the Distributed RDF Engines)

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 [51], Sesame [44] 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.

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

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 [11] 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 [28, 52, 10, 53] 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.

---

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

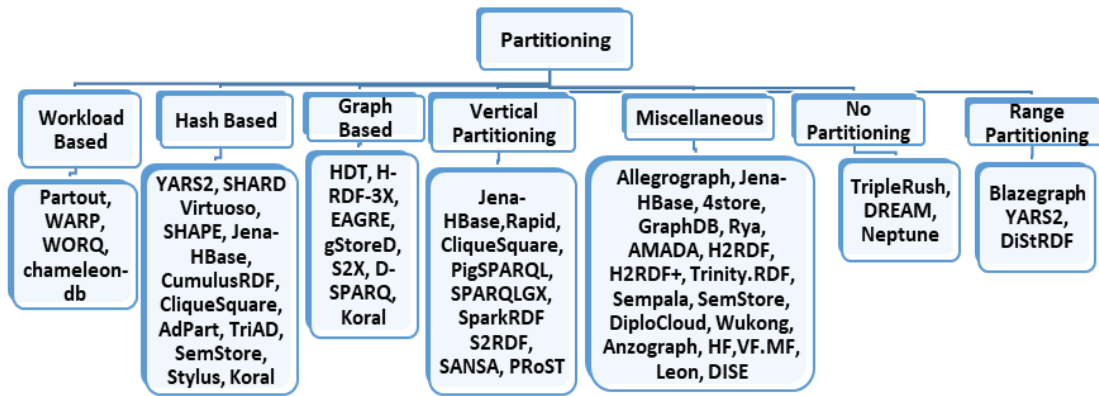



Fig. 11. Types of Partitioning used RDF Engines

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

- **Subject-hashed.** This technique assigns triples to partitions according to a the hash value computed on their *subjects* modulo the total number of required partitions (i.e., hash(subject) modulus total number of partitions) [54]. Thus, all the triples with the same subject are assigned to one partition. However, due to the modulo operation, this technique may result in high partitioning imbalance. In our motivating 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. Thus, a clear partitioning imbalance (3:1:7 triples) results.

- **Predicate-hashed.** This technique assigns triples to partitions according to a the hash value computed on their *predicates* modulo the total number of required partitions. Thus, all the triples with the same predicate are assigned to one partition. In our motivating example, 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 $p1$ will be assigned to first partition (red), $p2$ triples will be assigned to second partition (green), $p3$ triples will be assigned to third partition (blue), and $p4$ triples will be 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 [11].

- **URI Hierarchy-hashed:** This partitioning is inspired by two assumptions: (1) IRIs have path hierarchy, (2) IRIs with a common hierarchy prefix are often queried together [54]. This partitioning technique extracts path hierarchies from the IRIs and assigning triples having the same hierarchy prefixes into one partition. For instance, the extracted path hierarchy of "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" is "org/w3/www/1999/02/22-rdf-syntax-ns/type". Then, for each level in the path hierarchy (e. g., "org", "org/w3", "org/w3/www", ...) it computes the percentage of triples sharing a hierarchy prefix. If the percentage exceeds an empirically defined threshold and the number of prefixes is equal to or greater than the number of required partitions at any hierarchy level, then these prefixes are used for the hash-based partitioning on prefixes. In comparison to the subject-hash-based partition, this technique requires a higher computational effort to determine the IRI prefixes on which the hash is computed. In our motivating example given in Figure 12, 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 [11].

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.** Recursive bisection is a multilevel graph bisection algorithm aiming to solve the k-way graph partitioning problem as described in [52]. This algorithm consists of the following three phases: (1)) *Coarsening:* The initial phase is coarsening the graph, in which a sequence of smaller graphs $G_1, G_2, ..., G_m$ is generated from the input Graph $G_0 = (V_0, E_0)$ in such a way that $|V_0| > |V_1| > |V_2| >$

```
@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
```

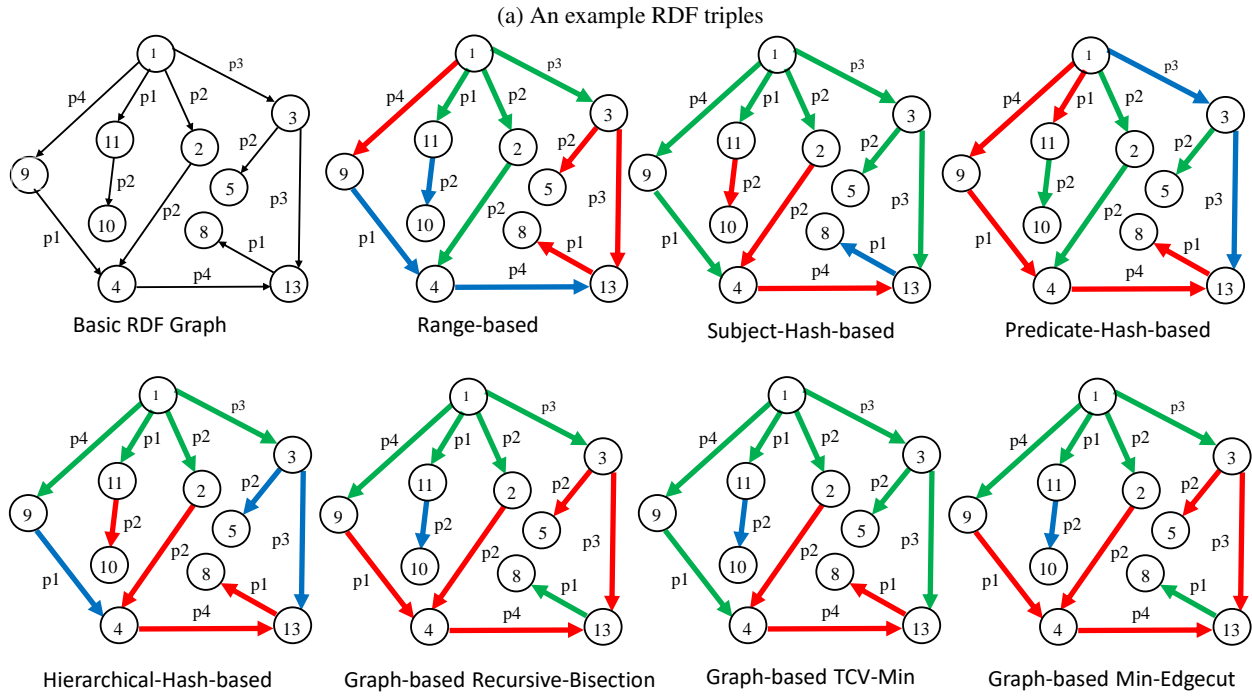
(a) An example RDF triples

(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.

... $> |V_m|$. (2) *Partitioning* In the second phase, computation of a 2-way partition P_m of the graph G_m takes place, such that V_m is split into two parts and each part contains half of the vertices. (3) *Uncoarsening* The third and last phase is uncoarsening the partitioned graph. In this phase the partition P_m of G_m is projected back to G_0 by passing through the intermediate partitions $P_{m-1}, P_{m-2}, ..., P_1, P_0$.

In our motivating example given in Figure 12, triples 1, 2, 4, 7, and 8 are assigned into green partition, triples 3, 5, 6, 9 and 10 are assigned into red partition, and triple 11 is assigned into blue partition.

- **TCV-Min Partitioning.** Similar to Recursive-Bisection, the TCV-Min also aims to solve the k-way graph partitioning problem. However, the objective of the partitioning is to minimize the *total communication volume* [55] of the partitioning. Thus, this technique also comprises the three main phases of the k-way graph partitioning. However, the objective of the second phase, i.e. the *Partitioning*, is the minimization of communication costs. In our motivating example given in Figure 12, triples 1, 2, 4, 5, 6, 8 and 9 are assigned into green partition, triples 3, 7 and 10 are assigned into red partition, and triple 11 is assigned into blue partition.

- **Min-Edgecut Partitioning.** The Min-Edgecut [52] also aims to solve the k-way graph partitioning problem. However, unlike TCV-Min, the objective is to partition the vertices by minimizing the number of edges connected to them. In our motivating example given in Figure 12, triples 1, 2, 4, 7 and 8 are assigned into green partition, triples 3, 5, 6, 9 and 10 are assigned into red partition, and only triple 11 is assigned into blue partition.

The graph-based partitioning techniques are computational complex, and may takes very strong 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

Redland [45] 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 stands for subject, P stands for predicate and O stands for object of the triple. Each hash is a 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 [51] 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 maintained for literals and resources/URIs. This scheme is very efficient in terms of storage because of the one-time storage of more than one occurrence 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 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.

---

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

[22]RASQAL library `http://librdf.org/rasqal/`

[23]https://jena.apache.org/documentation/tdb/architecture.html

[24]Jena: `https://jena.apache.org/`

[25]Jena ARQ: `https://jena.apache.org/documentation/query/`

[26]FUSEKI: `https://jena.apache.org/documentation/fuseki2/`

Engine	Storage					Indexing					Language			Query Processing				
	T	P	V	G	M	P	T	B	N	M	S	T	O	S	T	A	G	M
RedLand [45]	✓						✓				✓		✓			✓		
Jena1 [51]	✓							✓					✓					✓
TDB[18]	✓						✓				✓			✓				
Sesame [44]					✓		✓						✓			✓		
3store [56]	✓						✓						✓		✓			
Jena2 [33]	✓	✓					✓						✓					✓
Mulgara[19]	✓						✓						✓	✓				
RStar [57]	✓						✓						✓					✓
BRAHMS [58]	✓						✓						✓					✓
YARS [59]	✓						✓						✓	✓				
Kowari [60]	✓						✓						✓	✓				
RDF-Match [61]	✓						✓						✓					✓
SW-Store [38]			✓			✓						✓						✓
Hexastore [62]					✓		✓				✓							✓
RDFJoin[20]	✓						✓				✓							✓
Parliament [48]	✓								✓				✓					✓
DOGMA [40]				✓					✓				✓				✓	
Turbo$_{HOM++}$ [63]				✓		✓					✓						✓	
RDF-3X [64]	✓						✓				✓			✓				
RIQ [65]					✓				✓		✓							✓
Stardog[21]					✓		✓				✓							✓
gStore [39]				✓			✓				✓						✓	
Strabon [66]					✓				✓			✓		✓				
BitMat [67]					✓				✓		✓							✓
Triplebit [46]					✓				✓		✓							✓
DB2RDF [68]		✓						✓				✓			✓			
RDFox [69]					✓				✓		✓							✓
AMBER [41]				✓					✓		✓						✓	

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)

Sesame [44] 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 [56] 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 [33] 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.

---

[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 [57] 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 [58] main memory-based RDF engine. The RDF data is store in three hash tables – s-op, o-sp, p-so where s represents the subject, o represents the object and p represents the predicate of the triple. 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) [59] 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 maintains 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 [60] 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 maintained. Same like Mulgara, Kowari also maintains six indexes which are stored as 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 [61] 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 [38] 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 [70]. 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 [62] 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 –, one for each possible ordering of the subject s, predicate p, and object o. 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.

---

[29]http://mulgara.org/

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

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

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 joins.

Parliament [48] 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 [40] 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 sub-graph 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 [64] 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 [64] 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 maintained 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 [65] SPARQL query execution is based on *decrease-and-conquer* strategy. Rather than creating a single large index over the entire RDF graph, RIQ identifies groups of similar RDF graphs and creates small indexes on each group separately. A new vector representation has been used for RDF graphs along with locality sensitive hashing to construct the groups efficiently. RIQ creates a filtering index on the groups and compactly represents it as a combination of Bloom and Counting Bloom Filters. During query processing, RIQ employs a streamlined approach. The query plan is constructed by searching the stored index to quickly identify quickly identify candidate groups that may contain matches for the query. RIQ rewrites the original query to produce an optimized query for each candidate. The optimized queries are then executed using an existing SPARQL processor to generate the final results.

---

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

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 [39] 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 [71]. gStore uses two trees for indexing: (1) a height-balanced S-tree [72], 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 [73]. 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 maintains a two B+ tree two-column indexes and a B+ tree index on the id column respectively. Strabon has new extension of SPARQL query language, stSPARQL [73] 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 [67] 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 [46] stores RDF data in the form of bit matrix storage structure [67] 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 [68] 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 maintained 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 [69] 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 maintained 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

---

[33]https://www.stardog.com/docs/7.0.0/

[34]RocksDB: `https://rocksdb.org/`

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) [41] stores all of its RDF data in the form of multigraph. It maintains three different dictionaries of key value pairs namely a vertex dictionary, an edge-type dictionary and attribute dictionary. Same like three dictionaries, AMBER maintains 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[41] 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 [42]. 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[43], 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 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) [74] 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 [75] 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.

---

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

Engine	Storage					Indexing					Language			Query Processing					Partitioning						
	T	P	V	G	M	P	T	B	N	M	S	S	O	S	T	A	S	M	W	H	G	V	M	N	R
Allegrograph[35]				✓		✓					✓							✓					✓		
Blazegraph[36]				✓		✓					✓			✓											✓
YARS2[74]				✓		✓					✓							✓		✓					✓
SHARD[75]				✓				✓			✓							✓		✓					
4store[35]	✓					✓					✓			✓									✓		
Virtuoso[34]	✓					✓						✓			✓					✓					
HDT[37]			✓			✓					✓							✓			✓				
H-RDF-3X[3]			✓				✓				✓			✓		✓					✓				
GraphDB[38]				✓						✓	✓							✓					✓		
CumulusRDF[76]				✓		✓					✓					✓				✓					
Rapid+[77]		✓						✓				✓		✓								✓			
Jena-HBase[78]	✓	✓		✓		✓					✓							✓		✓		✓	✓		
Rya[79]				✓		✓					✓			✓	✓							✓			
AMADA[80]				✓		✓					✓		✓					✓				✓			
H2RDF[81]				✓		✓					✓			✓								✓			
SHAPE[82]			✓					✓			✓			✓		✓				✓					
WARP[7]			✓					✓			✓							✓	✓						
PigSPARQL[83]		✓							✓			✓		✓								✓			
EAGRE[42]				✓					✓		✓							✓			✓				
H2RDF+[43]				✓					✓		✓			✓								✓			
Trinity.RDF[47]			✓			✓					✓					✓						✓			
D-SPARQ[84]			✓			✓					✓			✓							✓				
CliqueSquare[85]		✓	✓					✓			✓						✓			✓	✓				
TripleRush[86]			✓						✓		✓						✓							✓	
chameleon-db[87]			✓						✓		✓						✓		✓						
Partout[6]			✓					✓			✓			✓					✓						
Sempala[9]		✓							✓			✓		✓	✓								✓		
TriAD[88]			✓				✓				✓			✓						✓					
SparkRDF[89]			✓						✓		✓					✓	✓					✓			
SemStore[90]			✓						✓		✓			✓								✓			
DREAM[5]			✓					✓			✓			✓										✓	
DiploCloud[91]				✓					✓		✓												✓		
SPARQLGX[83]		✓				✓						✓		✓								✓			
S2RDF[10]		✓				✓						✓		✓								✓			
AdPart[92]				✓		✓					✓			✓						✓					
S2X[93]			✓						✓		✓					✓						✓			
gStoreD[8]				✓					✓		✓					✓						✓			
Wukong[94]			✓						✓		✓							✓					✓		
SANSA[95]		✓				✓						✓			✓							✓			
Stylus[96]				✓					✓		✓							✓		✓					
Koral[97]			✓			✓					✓		✓					✓		✓	✓				
PRoST[98]		✓				✓					✓			✓								✓			
WORQ[99]		✓							✓		✓							✓	✓						
Anzograph[39]			✓						✓		✓							✓					✓		
Neptune[40]			✓			✓					✓		✓	✓										✓	
HF,VF,MF[100]				✓					✓		✓					✓							✓		
DiStRDF[101]	✓	✓										✓						✓							✓
Leon[102]				✓					✓			✓						✓					✓		
DISE[103]				✓		✓						✓						✓					✓		

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)

4store [35] 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) [104] 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 [105], using a 4-bit radix. The query engine is largely based on Relational Algebra. The join ordering is based on the cardinality estimations.

Virtuoso [34] 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 maintains a fewer number of indexes. The gspo is the default index used in the Virtuoso. In addition, it maintains 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.

HDT Header (metadata), Dictionary (strings occurring in the dataset), and Triples (graph structure) (HDT)[44] is a form of binary representation to achieve high compression and indexing for efficient querying. HDT is used to compress massive RDF datasets. The header contains metadata, and the dictionary contains the catalogue of all the resources, literals, and blank nodes. A triple component represents the Graph in the tuple of three components Subject, Predicate, and Object. In **HDT**, one can take advantage of fast search operations to different parts of the file without having to decompress it fully. HDT works on the principle of in-memory access, and the whole file is uploaded to memory to worked upon as a data structure. In-memory operations save the expensive indexing. All this process occurs at a server-side, and it is only restricted to an only containing **SPO, SP?, S?? and ???** queries. To be able to work on the client-side and for queries containing other triple patterns, HDT includes two indexes to the original data structure of HDT. The original **Sp** based index is now loaded as a wavelet tree despite an array. Another index called **O-Index** is added with the triples. This new representation is called **HDT-FoQ: HDT Focused on Querying**.

The triple portion of the **HDT** contains the triples in a binary format in a tree form. The tree consists of three levels with Subject as a root node followed by Predicate and Object. Query execution in **HDT** starts from the listed Subject as a root node in the SPARQL query. For queries starting with Predicate and Object, HDT has the new representation of its data structure after the addition of two indexes. The new representation is **HDT-FoQ: HDT Focused on Querying**. HDT-FoQ only supports conjunctive queries with index and merge joins and still lacks the support for dis-junctions, and more operators.

H-RDF-3X [3] uses RDF-3X as its storage component to RDF data as graph. It also makes use of a k-hop guarantee to replicate vertices and edges, which are k-hop away. RDF-3X installed in every partition contains duplicated values because of replication. H-RDF-3X makes use of RDF-3X internal statistics to generate efficient query plans. Because of replication and RDF-3X's full scans of the dataset in a partition, it may cause storage and time overhead. H-RDF-3X also utilizes indexes, which are built over all possible permutations of Subjects, Predicates, and Objects in a partition. SPARQL query processing in **H-RDF-3X** is a decision between two things. Either a query will be Parallelizable Without Communication (PWOC) or not. If a query can be answerable without involving expensive Hadoop jobs, then the query is PWOC. This can be achieved through hop guarantee, larger the replication larger is a hop guarantee. That will help to avoid Hadoop in some queries. If the query is not PWOC, then it will be decomposed into subqueries that are going to join by expensive Hadoop jobs. Hadoop jobs are directly proportional to the number of subqueries. In order to avoid expensive Hadoop jobs, heuristics are used to minimize the number of subqueries.

GraphDB stores its all of data, indexes, entity pools in the directory called storage[45]. GraphDB is packaged as Storage and Inference Layer (SAIL) for the Sesame RDF framework, and RDF4J API's for storage and querying. **GraphDB** contains many automatic indexes apply to implicit or explicit triples. Two main indexes are SPO, OSP, and POSIndexes in GraphDB offers many advantages for specific datasets, retrieval patterns and query loads. For SPARQL query evaluation, the **GraphDB** makes use of two indexes. GraphDB makes use of predicate object, subject (POS) and the predicate, subject, object (PSO) index to perform query evaluation. Another index, known as Literal index may be used during query evaluation if the query or subquery contains filters with literals constraints using comparison or equality.

CumulusRDF [76] employs a NoSQL system, Apache Cassandra for implementation of its storage layouts, i.e., hierarchical and flat layout. Cassandra uses a set of distributed hash tables in which each hash working as a hash key is responsible for storing rows in a particular range of hash keys. These layouts help in linked data and basic triple pattern lookup. The hierarchical layout makes use of super columns of Cassandra. Three permutations of s,p,o are maintained to enable fast lookup for triple patterns and named graphs. CumulusRDF also employs a secondary index, cspo to map the column values to row keys. For query processing, Sesame processor translates the queries into index lookups of Cassandra.

RAPID+ [77] is a Pig-based system that uses a Vertical Partitioning storage strategy to save RDF data. To store data in VP tables fashion in Pig, a *SPLIT* command of Pig is used. In previous approaches data transfer between the mappers and reducers using Map Reduce frameworks show high communication and I/O costs. To minimize these costs, Nested

---

[44]http://www.rdfhdt.org/hdt-internals/

[45]http://graphdb.ontotext.com/documentation/6.6/standard/storage.html#graphdb-s-indexing-options

Triple Group Algebra (NTGA), new intermediate algebra comes into action for the optimization of a query process. The main concept of NTGA is based on the TripleGroup, which cluster of triples with same subject or object like in Star-shaped queries.

Jena-HBase [78] a distributed framework for scalable storage and querying for massive RDF data. It supports multiple storage layouts of Vertical Partitioned, Indexed, Hybrid and Hash over several HBase tables. Through these variety of storage layouts, there is a tradeoff between storage and query performance in Jena-HBase. Experimental evaluation shows the superiority of Hybrid layout due to combination of other layouts. For querying, Jena-HBase has query runner block that provides APIs for different storage layouts.

Rya [79] employs Accumulo, a key-value and column-oriented store as its storage backend. Rya can also use any other NoSQL database as its storage component. The row-id part stores all the key-value pairs. Rya stores three index permutations, i.e., (spo, pos, and osp). In Rya, OpenRDF Sesame SAIL APIs are used for storage and querying purpose. Specifically, index nested loop joins are used for SPARQL query execution using MapReduce fashion. The count of 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. This updation is critical because the query planner will use this information to reorder the query according to the triple pattern with the highest selectivity.

AMADA [80] is implemented on Amazon Web Services (AWS) cloud infrastructure. This infrastructure supports storage, indexing and querying as software as a service (SAAS). AMADA builds indexes using SimpleDB, which supports SQL queries. SimpleDB is based on key-value storage solution. It supports no joins, single relation queries. 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 [81] is a kind of distributed RDF system that stores data as a multiple indexing scheme over HBase in combination with Hadoop framework. HBase is a horizontally scalable NoSQL store. In H2RDF, three permutations of s,p,o (spo, pos, and osp) are maintained. These permutations are maintained over the HBase tables in the form of key-value pairs. For query processing, join planner algorithm in H2RDF makes use of statistical information collected during data loading. The join planner algorithm takes into account the selectivity information, execution cost and greedily selects the join for execution. The join executor module of H2RDF decides between the execution of joins in a centralized and distributed manner.

Semantic Hash Partitioning-Enabled distributed RDF data management system (SHAPE) [82] uses RDF-3X to store RDF triples across nodes of a cluster. Storage of triples is done by semantic hash partitioning based on the URI hierarchy to achieve data locality. In the URI hierarchy, triples with the same Subject or Object are identified and are placed in the same partition. K-hop expansion is applied, which replicates the triples on the border to achieve data locality. RDF-3X offers different statistics for efficient query plans, but a huge TT also suffers from a lot of self joins. Replication of triples on a border is cumbersome for datasets because of full semantic hash partitioning. URI based hierarchy is problematic is some datasets where URI's of both datasets are uniform. **RDF-3X** in **SHAPE** maintain indexes covering all possible permutations of Subject, Predicate, and Object, which are stored as clustered B+ trees. Replication and extensive indexing of the entire dataset at each node cause extra storage overhead. In SHAPE at the time of grouping on the base of similar Subject or Object and assigned to the partition **indexed** by the hash value of their subject or object value. SPARQL query execution in **SHAPE** is divided into three phases, i.e., query analysis, query decomposition, and generating distributed query execution plans. Query analysis is performed to term query as intra partition or inter partition. If a query is inter partition, then it is split into sub-queries by query decomposer to be executed by client machines. Intermediate results from different client machines are loaded into HDFS and joined using MapReduce Hadoop joins using subsequent Map and Reduce phases.

Workload-Aware Replication and Partitioning **(WARP)** [7] also uses RDF-3X to store triples in partitions among a cluster of machines. Storage of triples is done after taking into consideration advanced replication methods and typical query workload. The replication method replicates the vertices and edges occurring at a border of partition by using the n-hop guarantee method. Query workload consists of a set of queries that are aggregated from different interfaces that contain queries having the same structure. WARP also uses the underlying RDF-3X indexes. **WARP** uses a query workload to process SPARQL queries efficiently. 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 [83] also uses VP for storing triples. **PigSPARQL** does not maintain a costly index structure. Because of this characteristic, PigSPARQL is well suited for "Extract, Transform, Load" like scenarios. The main component of **PigSPARQL** is a translation of SPARQL queries into Pig Latin [13] scripts on Apache Pig. The translation follows

a common principle, which 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 translated into the SPARQL algebra tree. In **PigSPARQL**, during the stage of translation of SPARQL to Pig Latin, in the first step, an abstract syntax tree is generated from the SPARQL query using the Jena ARQ framework. The abstract syntax tree is translated into the SPARQL algebra tree. At this optimization for filters and rearrangement of triples, patterns based on selectivity are also 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. Optimizations in the translation process employ multi join to reduce the number of joins in Pig Latin. Multi joins can also be used where many consecutive joins refer to the same variable.

*E*ntity *A*ware *G*raph comp*RE*ssion technique (**EAGRE**) [42] stores RDF data in HDFS in a (key, value) fashion to preserve both the semantic and structural information. After uploading data to the HDFS and extract "entities" and "entity classes" from the original RDF graph G and build the "compressed RDF entity graph, novel RDF representation model in the (key, value) store so that the inner correlation of RDF data remains valid. Due to the huge volume of RDF data, MapReduce is used to perform the process of extractions. After the extraction of entities and entity classes, the RDF data layout comes into place. RDF data layout encompasses two things one is which RDF data should be deployed to which of the slave machines, and the other is how this data should be organized. For the first purpose **Metis** [106] is used to partition the entity graph among slave machines. For the purpose of the organization, which also serves as an indexing structure, **EAGRE** adopts a space-filling curve technique, which is used to index high dimensional data. The space-filling curve offers three types of advantages. First is that it provides an efficient solution to save the I/O cost of unnecessary disk scans. Second is the order-preserving property of it. Thirdly it functions well in case of data updates, which is not very efficient in modern distributed RDF engines. The distributed and centralized manner through MapReduce and over a single node, respectively. SPARQL query processing in **EAGRE** is all about minimizing I/O costs by efficient distributed scheduling approaches. All this is done to reduce the reading of data blocks which are to be read for query evaluation.

H2RDF+ [43] uses HBase[46] table to store RDF data. Two HBase tables are also used to store dictionaries to translate string values to IDs and vice versa. HBase tables are also used to store indexes for data stored in HBase. The main component of **H2RDF+** is a distributed indexing scheme to load and index large RDF datasets. H2RDF+ also keeps statistics for the aggregated index to use for estimation of pattern selectivity, join output, and join cost. H2RDF+ maintains a multi-way merge join and sort-merge join algorithms for sorted and unsorted data respectively to apply on our distributed index or intermediate results. Both merge algorithms can be executed in a distributed I/O is done to postpone the expensive MapReduce jobs to minimize the query evaluation time.

SPARQL query execution in **H2RDF+** makes the decision of the execution order of the different joins to minimize query execution time. More joins make the choice of join order makes all computation expensive. For the facilitation of the execution of join in every step, greedy, cost-based, and online planner is used. This join cost model makes use of stored statistics in HBase tables and also helps in making the decision of query execution in a distributed or centralized manner.

Trinity.RDF [47] store RDF data in its native graph form. Trinity.RDF is based on Trinity key-value store. In Trinity.RDF RDF entity is termed as a graph node, given a unique id as a key and adjacency list with incoming and outgoing edges as a value. The adjacency list contains predicate and node id of the connected nodes with predicate as a label on the edge. Through this storage layout, one can easily find any node id of connected nodes by giving any node. In Trinity.RDF, RDF graph is partitioned randomly across by applying hashing on nodes. This will retrieve the key-value pair of that node-id. This structure helps us to explore the graph from any node through accessing its adjacency list. **Trinity.RDF** maintains local predicate and global predicate indexing. Local predicate index is applied on an adjacency list, which contains predicate and node-id. All pairs of them are sorted first by predicate wise and then node-id wise, apart from that aggregated index containing the node with given predicate and number of connected nodes by the predicate.

The exploration-based approach employed in **Trinity.RDF** avoids the generation of unnecessary intermediate results. In first step of SPARQL query processing, query *Q* is decomposed into subqueries *qi* to *qn*. Then matches for each *q* are found, from these matches graph is explored to find matches for connected nodes. All of this process is performed in parallel in each slave machine. In the last stage, the gathering of all single triple patterns at a centralized query, the processor is merged to produce the final result.

D-SPARQ [84] is a distributed RDF engine, which is implemented on top of MongoDB, a NoSQL document database. D-SPARQ takes RDF triples as input and makes graph. The graph is then partitioned among nodes of a cluster. After partitioning, the triples with the same subject as vertex are placed in the same partition as the vertex. In

---

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

addition, a partial data replication is then applied where some of the triples are replicated across different partitions for parallelization execution of queries. Grouping the triples based on the same subject enables D-SPARQ to efficiently execute star patterns queries. D-SPARQ also uses indexes which 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 [85] is a Hadoop-based RDF engine used to store and process massive RDF datasets. It stores RDF data on the basis of semantic hash partitioning in order to minimize the number of MapReduce jobs and amount of data transfer among cluster nodes. In CliqueSquare, the objective of partitioning is to perform called as co-located or partitioned joins. In these join RDF data is placed in such a manner that the maximum number of joins are evaluated in map phase. CliqueSquare stores RDF data in three replicas. In order to apply SPARQL queries on RDF dataset, the CliqueSquare uses a clique-based algorithm, which produces query plans that minimize the number of MapReduce stages. This algorithm 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 [86] is built upon the Signal/Collect [107], a parallel and distributed graph processing framework written in Scala, similar to GraphX. In TripleRush, three types of vertices exit. The index vertex corresponds to a triple pattern, triple vertex is RDF triple and query vertexes coordinate the query execution. In TripleRush, index graph is formed by index and triple vertexes. 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. In TripleRush there is no partitioning performed.

chameleon-db [87] stores its data and queries in a graph form. It is based on Workload aware partitioning mechanism to adjust storage layout to efficiently execute queries. chameleon-db indexes the partitions by workload aware partitioning indexing technique. It is an incremental indexing technique which uses a decision tree to keep track of relevant segments of the queries. It also uses vertex index which is a kind of hash table to contain URIs of vertices in subset of partitions. For keeping the track of minimum and maximum literal values, chameleon db also contains range index. For SPARQL query execution, the chameleon-db performs the subgraph matching. The chameleon-db performs the RDF graph partitioning by partition-restricted evaluation (PRE). This technique is used to filter the dormant triples which don't contribute to the final result.

Partout [6] also uses RDF-3X for storing its triples by making use of Workload aware partitioning. It stores triple based on the query workload. It makes partitions based on graph patterns occurring together in a query and assigns partition of data to a cluster node running RDF-3X to avoid inter-partition communication. **RDF-3X** in **Partout** maintain indexes covering all possible permutations of Subject, Predicate, and Object, which are stored as clustered B+ trees. Replication and extensive indexing of the entire dataset at each node cause extra storage overhead.

In **Partout** SPARQL query is issued at a server which generates suitable query plans for distributed query execution. The query execution process is passed through different steps. The query is issued at a server having no direct access to original data. The server utilizes a global statistics file, which is generated a time of making partitions containing information about partitions definition, size, and mapping to a client. These statistics are used to convert SPARQL queries according to an RDF-3X execution plan, which resembles like leaves of a tree with access to data. In the next step, 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 triple's locations in a cluster setup. The refined query plan is executed by client machines in parallel, and the final result is joined by Binary Merge Union (BMU).

Sempala [9] a columnar storage format for Hadoop known as **Parquet**[47] for storing RDF triples. Advantages of Parquet are that in column format, all columns are stored consecutively on a disk. This feature enables better compression and encoding, as all data is of the same type. The parquet does not store null values. Null values are determined by the definition levels. The Parquet is designed for supporting a wide table. Because of this characteristic, Sempala uses a single Unified Property Table consisting of all properties for the reduction of self joins. This results in a zero number of joins for Star pattern queries. Sempala's Unified Property Tables lack the support for null values and multi-valued attributes. Null values are not explicitly stored in Parquet. Multi-valued attributed cause little bit storage overhead. To tackle the issue of multi-valued, Parquet uses nested data structures. As Parquet is used with Apache Impala [108], Impala at the time of Sempala doesn't support nested data structures. For storing multi-valued attributes, duplication of rows with different values is done. The storage of multi-valued attributes causes enormous storage overhead when there exist many multi-valued attributes. The run-length encoding of Parquet mitigates this effect.

Sempala has no indexing use in its execution of queries. In **Sempala** main component is a translation of SPARQL queries into SQL for running on Apache Impala. In **Sempala**, the main component of query execution is the translation

---
[47]http://parquet.apache.org

of SPARQL to SQL. This translation is done by the query compiler of Sempala. Like other translation approaches, Jena ARQ is used to generate an equivalent algebraic tree of the SPARQL query. Then this tree is parsed in a bottom-up fashion after applying some optimizations like filters. Parsing generates the Impala SQL expressions based on Unified PTs to be executed.

Triple Asynchronous and Distributed (TriAD) [88] stores triples after the process of graph summarization. In the process of graph summarization, the summary graph is formed, which contains only relevant triples with respect to a query. In this process, the original large data graph is converted into smaller graphs that retain the major characteristics of an original data graph. The complex query applied against the summary graph is faster than applying on the original RDF data graph. **TriAD** being a master-slave architecture, maintains indexes at both master and slave machines. After distributing the summary graph among partitions, it is also indexed at a master site. The summary graph is indexed in the adjacency list format in large in-memory vectors, the permutations of Predicate, Subject, and Object (pso) and Predicate, Object and Subject (pos) of the summary graph as hash partitioned shards in a lexicographic order to support exploratory search. The slave machines create six in-memory vectors of triples received from the master machine. These are divided into two parts, one is Subject-based, and the other is Object-based. Subject-based are (spo, sop, pso), and object-based are (osp, ops, pos).

In **TriAD** for query execution, SPARQL query is parsed and translated into a query graph and passed through two stages. In the first stage, pruning of data is done through an exploratory algorithm for join ahead pruning at the later stages. This process is done with the help of summary graph statistics maintained at the master machine. The second step involves the processing of queries for data distributed among slave machines. In this step query processor selects the best join order by dynamic programming (dp) optimizer and distributed cost model. Each slave machine receives a global query plan and pruning information from the master machine. Based on this information, slave machines perform multiple join operators in parallel, which share their results by asynchronously sending and receiving to/from other slaves. When query processing finishes, intermediate results from slaves are merged at the master node.

SparkRDF [89] 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 cut down search space and memory overhead. There are five kinds of indexes created in SparkRDF, i.e., C (Class subgraphs), R (Relation subgraphs), CR, RC and CRC (combinations of class and relation subgraphs). These indexes are modeled as RDSGs (Resilient Discreted SubGraphs). For query processing, SparkRDF employs Spark APIs and an iterative join operation to minimize the intermediate results to perform subgraph matching. The final subgraph is obtained by the computation of matching every triple pattern in a query.

SemStore [90] uses a centralized RDF engine *Triplebit* as its storage component. SemStore partitions data in a novel way of Rooted Sub-Graph (RSG), which is a way to localize the shapes of all queries. It implements hash function to assign RSG 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. SemStore follows a master-slave architecture having one master node and many slave nodes. After partition, the data partitioner maintains 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 [5] uses RDF-3X single node RDF engine as its storage component. Generally distributed RDF engines partition RDF dataset into different subgroups, but DREAM replicates an entire dataset on every node with RDF-3X installed. This replication causes storage overhead but avoids inter partition communication among cluster nodes. This replication results in non-shuffling of intermediate data and only identifiers of triples are communicated. RDF-3X installed on every node helps to achieve statistics for query evaluation. **RDF-3X** in **DREAM** maintain indexes covering all possible permutations of Subject, Predicate, and Object, which are stored as clustered B+ trees. Replication and extensive indexing of the entire dataset at each node cause extra storage overhead.

In the query execution phase of **DREAM**, SPARQL queries are partitioned instead of dataset. In the first step, the SPARQL query is translated into a directed graph. Then the query planner partitions the directed graph into multiple sub-graphs. Query planner locates the vertices with certain degrees as a join vertex, i.e., degree greater than 1 is join vertex (JV). At this step, empty sets are created for join vertices are created and populated with sub-graphs of the original directed graph using a Rule bases strategy. For every join vertex, only one set is selected and assigned to different nodes of a cluster. Now all sets can be run in parallel and can exchange necessary information. For joining intermediate results to get a final result, a hash-based join algorithm like in relational query optimizers is used.

DiploCloud [91] makes use of three things as its storage components on single node system *dipLODocus* [109]. A molecule clusters, which extend property tables to form RDF subgraphs that groups related URIs in a nested hash tables.

Literals are stored in Template lists as in a columnar database system. For storing URIs indexes, a molecule index is used. 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 query execution, the master node distributes the subqueries among slave nodes, which runs subqueries and return intermediate results to a master node. In the case of distributed joins, if the intermediate results are small, then master node performs the joins. In other case distributed hash join is performed. DiploCloud makes use of different partitioning strategies like Scope-k Molecules, Manual Partitioning and Adaptive Partitioning to distribute data among cluster.

SPARQLGX [110] stores its RDF data in Hadoop Distributed File System (HDFS) [111] in VP fashion. SPARQLGX achieves three advantages of fast conversion, compression, and slight indexation by saving data in VP. Fast conversion is done because of linear and traverses dataset only for a single time. RDF data is compressed because of the removal of Predicate from RDF triple. In **SPARQLGX**, a minor form of indexation is achieved because of saving different groups of triples by their predicate. Translation of SPARQL query triple patterns into SPARK code contain some steps in **SPARQLGX**. In a first step, concerned data from the disk is accessed against each triple pattern and then aggregated. To translate the conjunction of triple patterns, they are joined on the base of their common variables as a key: **keyBy** in Spark. In the case when there is no common variable cartesian product is used to join triple patterns. In the final step after translation, a **map** is used to retain only desired variables and, at this stage, further optimizations, i.e., removing duplication takes place. This step-up also translates keywords like **UNION** and **OPTIONAL** located in a **WHERE** clause. Query execution of **SPARQLGX** has two options, one which involves statistics for optimization as triple patterns are joined on the bases of their common variables. Minimum intermediate results produced in this process will result in minimum communication among processing nodes to achieve faster execution. Statistics on these intermediate results are used for optimization. Another option for query planning is direct when there is a single query or update query. This scenario requires minimum query preprocessing and evaluation time. To achieve this original triple source file, not predicate based storage is taken, and another translation process is the same. This option is called "direct evaluator" or SDE.

S2RDF [10] is an RDF engine which indirectly runs on the top of Spark [112] by translating SPARQL queries into SQL. For storing triples, S2RDF uses VP to reduce the size of input data. To avoid skewness in data and because of that dangling tuples cause unnecessary Input/Output and comparison's in joins execution in limited in-memory of Spark, an extended version of VP is implemented. This extension is known as Extended Vertical Partitioning (ExtVP) related to Join indexes [24]. ExtVP is based on the position of a variable that occurs in both triple patterns (called join variable) determines the columns on which the corresponding VP tables must be joined. We call the co-occurrence of a variable in two triple patterns a **correlation** i.e. subject-subject correlation (SS), subject-object correlation (SO), object-subject (OS) and object-object correlation (OO). Semi joins reductions for correlations except (OO) is precomputed. These precomputations are not mandatory. ExtVP has the advantage of choosing VP instead of ExtVP tables when the difference between both in terms of size is not large enough. This is done through an optional selectivity threshold.

S2RDF slight indexation which reduces response time in some cases since predicates classify triples. In terms of index updation, insertions are quickly done by appending new triples to ExtVP tables. Deletions are a little bit complicated as it will need to remove Predicate from every ExtVP table, which may result in dangling tuples. In the current implementation of S2RDF updation cannot be implemented because of the immutable nature of HDFS. **S2RDF** does the translation of SPARQL into SQL by parsing the SPARQL query parsed into algebra tree and through Jena ARQ[48], and then optimizations like a filter, etc. are applied. Then this tree is traversed from bottom up to generate corresponding single SparkSQL query.

For query execution in **S2RDF**, the order of triple patterns in equivalent query expression governs query performance. S2RDF orders triple patterns on the basis of selectivity **i.e. sel(tpi) less then sel(tpi+1)**. In query optimization, patterns with more bound values are selected, and cross products are avoided as they have the worst selectivity.

AdPart [92] is a kind of specialized distributed RDF engine that follows typical Master-Slave architecture. In terms of storage, AdPart slave machines store their local set of triples *Di* in an in-memory data structure to support predicate based search operations.

1. if Predicate P is given, return will be $\left\{ (s,o) \middle| \langle s,p,o \rangle \in Di \right\}$.

2. if Subject S and Predicate P is given, return will be $\left\{ o \middle| \langle s,p,o \rangle \in Di \right\}$.

3. if Object O and Predicate P is given, return will be $\left\{ s \middle| \langle s,p,o \rangle \in Di \right\}$.

For all the above searches, the presence of predicate P is necessary.

---

[48]https://jena.apache.org/documentation/query/

For all the searches involving Predicate P **AdPart,** performs hashing of triples in each slave machine by the predicate P. Resulting index or simply P-index can support searches by the input of P. Further two more hashing of Predicate-Subject (ps-index) and Predicate-Object (po-index) is maintained at each slave machine for 2 and 3 search. Each slave machine also contains an in-memory Replica Index for adaptivity, and it is grown incrementally after query workloads. It is maintained to replicate data that is accessed by most of the queries, and that is indexed in a structure called Pattern Index (PI) at the master machine. That data is replicated among all slave machines.

In **AdPart**, for SPARQL query execution, each slave machine also keeps a query processor that works in two modes. One is *distributed mode* and other is *parallel mode*. Query planning involves the use of global statistics from the statistics manager and the pattern index from the redistribution controller to decide if a query can be processed with or without communication among nodes of a cluster. Each node independently executes the queries which are don't require communication. For the other case, the query planner uses dynamic programming based cost optimization to produce an efficient query plan. Distributed semi-join algorithm (DSJ) is used for communication among nodes of a cluster.

S2X [93] RDF data has an inherent graph-like structure. S2X makes use of this property to process SPARQL queries on the top of the Spark component called GraphX [113]. S2X store triples among different worker machines after applying hash-based encoding on Subject and then Objects through GraphX default partitioner. S2X converts the RDF graph into a property graph data model of GraphX. The property graph is a directed multigraph having vertices and edges representing nodes and links, respectively. GraphX maintains the structure and properties of the graph separately to help in preserving the structure of the graph while properties are being changed. **S2X** has no indexing in its query execution path.

For query processing, SPARQL in **S2X** combines matching of graph patterns with relational style operators to produce solution mappings. Matching of graph patterns is started by distributing all triple patterns to all vertices. Then matching of an edge is done with the predicate of the triple. If a match is found, then this information is passed to direct neighbours by exchanging messages. Through this, partial results are merged. Relational operators of SPARQL are implemented through Spark API, e.g., In S2X OPTIONAL keyword is applied through the left-outer join.

gStoreD [8] is based on the strategy of partitioning the data graph but not decomposing the query. Overall gStoreD is based on the partial evaluation and assembly framework. gStoreD stores the triples by modifying the centralized RDF engine gStore. gStore stores RDF triples in an adjacency list table and encodes them into a bitstring also known as vertex signature. In **gStoreD** there is no indexing mechanism. Only in order to compare fairly with Trinity.RDF and TriAD whole RDF graph together with the corresponding index into memory. Query processing in **gStoreD** encompasses different steps. In the first step, the SPARQL query is issued against each site. Every site finds the local matches against the applied query by the use of partial evaluation. In the last step, all the local matches are assembled for the computation of cross matches. For assembly, two strategies may be used. One is a centralized assembly, and the other is distributed assembly. In centralized assembly, a single site is used to assemble all the local matches. In distributed assembly, local matches are joined in parallel at different sites. The work in [114] proposes some optimizations for partial evaluation and assembly for pruning useless intermediate results.

Wukong [94] stores its RDF data in a directed graph form on a distributed key-value store. Wukong is based on two kinds of indexes. One is normal vertex which refers to a subject and object and other is predicate index which maintain all subjects and objects labeled with the particular predicate using its in and out edges respectively. Wukong makes use of Remote Direct Memory Access (RDMA) for query distribution. Each query is divided into multiple subqueries on the basis of 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)** [95], a data processing engine contains different layers for large scale analysis of RDF data. **Sparklify** [49] is one of a layer in the SANSA framework and serves as a default query engine for SPARQL-to-SQL translation of SPARQL queries into Apache Spark code through Spark SQL. Sparklify makes use of extended Vertical partitioning (VP) of RDF data. Sparklify stores data in Hadoop Distributed File-System (HDFS) in order to be efficiently read by Spark. SANSA stack achieves a minor form of indexation, because of saving different groups of triples by their predicate.

Sparklify in SANSA stack translates the SPARQL queries to SQL to execute as a Spark code. Sparklify supports R2RML and Sparqlification Mapping Language (SML). Sparklify translates a SPARQL query into a SQL query and a set of SPARQL result variable definitions. The line of action of Sparklify contains similar steps like other translation schemes. In the first step, algebra is created, and in the other step, optimizations are applied. The query mechanism of **SANSA Stack** is dependent upon the efficient translation of SPARQL queries into SQL. The resultant queries are then applied to the Spark SQL engine. The result of this operation is a data frame which is then mapped into a SPARQL binding.

The storage scheme of the **Stylus** [96] is built upon a key-value store. Specifically, Microsoft Trinity Graph Engine is used. In order to enable easy searching of subjects by the given object, Stylus maintains a reverse triple. As it is obvious in many datasets that more than 90 percent of entities are represented by only a small number of predicates. This fact causes to form a template of similar entities. Based on this reason, Stylus uses a data structure called xUDT to represent such "templates" – combinations of predicates. In **Stylus** storage by triples and reverse triples in the xUDT data structure makes retrieval faster. An index in Stylus maps a given predicate to xUDT containing a matched predicate. Through this Stylus enumerate the subjects and objects related to a specific predicate. The storage scheme of Stylus also supports data updates. Updates in Stylus is performed without disturbing the original schema.

In **Stylus**, for the purpose of matching the SPARQL queries are divided into two types. The first type is used for matching against preprocessed data which also includes indexes. And the other type is matched against intermediate results. For preprocessed data, Stylus uses an optimized query planner, and for intermediate results, set operations are used. Queries are represented as a Twig, which is the height of two. Stylus uses xTwig, which is an extended version of the Twig to postpone cartesian products.

Koral [97] stores the RDF data in a graph form using hash partitioning. Due to the huge size of the RDF graph, this graph needs to be minimized. In order to minimize the graph, the dictionary is maintained, which contain the mapping of resources to ids. Koral follows the master-slave architecture in which slaves, after receiving the chunk of the graph, creates spo, osp, and pos indexes. Koral is a distributed system which offers the facility of alternate approaches for each component in distributed query processing. So there is no specific language used in the respective paper, as different components have different query language interface. For query execution, Koral offers different variations in two things. These two things are the additional information which is added the different chunks of the graph. And the other thing is the difference of creation of query execution tree made for different queries.

PRoST [98] stores data twice using VP and PT. Two different storage strategies are employed to take advantage of both according to different query types. VP in PRoST has some advantages and disadvantages, like in SPARQLGX. A PT consists of a unique table where each row contains a distinct subject and all the object values for that subject, stored in columns identified by the property to which they belong. PT has an advantage of avoidance of several joins when triple patterns in a query share the same subject, i.e., Star queries. PT suffers from the problems of the number of nulls and multi-valued attributes. A large number of nulls occur because not every subject-predicate pair have an object. To tackle the problem of nulls, Parquet uses a run-length encoding. Overhead by multi-valued is negligible compared with the execution of one join in a query. **PRoST** offers no indexing in its execution path.

In **PRoST**, SPARQL queries are translated into Join Tree format. In Join Tree format, every node represents the VP table or PT's subquery's patterns. Join tree takes conjunction of triple patterns are considered. Triple patterns having the same subject are grouped together and translated into a single node with a special label, denoting that we should use the PT. All the other groups with a single triple pattern are translated to nodes that will use the VP tables. The presence of a joining edge between every pair of nodes sharing a variable is the possibility of more than one join tree.

WORQ [99] performs the partition of RDF data in online fashion. WORK uses a query workload driven approach during the partitioning of data stored in a Vertical partitioning manner. In each of the SPARQL query, WORQ considers each triple pattern as a subquery. After this, the partition is performed by the join attribute of each subquery. Indexing in **WORQ** is challenging in case of queries with unbound properties. In this case, any other storage scheme is not preferred. In order to process queries with unbounded properties, WORQ uses Bloom Filters as indexes over the Vertically partitioned tables. WORQ performs two steps, i.e. identification and verification, to perform the indexing.

SPARQL query processing in **WORQ** is dependent on many steps. In the first step, based on the query workload, partitions of data are performed. Then online reductions of RDF joins are performed by bloom filters. Caching of RDF joins reductions is performed to boost the query performance. At this point, the efficient performance of identification and verification steps is important. At first, the unbound and bound attributes are identified. As the bloom filter also incurs false positives, a verification step is also performed by applying a filter related to abound attributes with the value indicated in the query triple pattern.

Anzograph[49] stores data in a graph form and supports massive parallel processing. Anzograph can be installed on single or multiple nodes. Users of Anzograph are not required to create and maintain partitions, it's automatically handled. There are no indexes created or maintained in **Anzograph**. Anzograph handles all itself for the users[50]. **Anzograph** makes use of SPARQL 1.1 queries to send to receive over HTTP. Developers can also make use of standard graph APIs to use Anzograph in their products. Query processing in **Anzograph** follows a master-slave architecture. The query is issued at a master node, which passes the query through parsing and planning. Planner decides the type of join or an aggregation needed for a query. Then the code generator unites the different steps into segments. Then all

---

[49]https://docs.cambridgesemantics.com/anzograph/userdoc/features.htm
[50]https://www.cambridgesemantics.com/anzograph-db-benchmarking-guide/

the segments are packaged for query into a stream. The master sends this stream to all the nodes, and after processing parallely, all the nodes send their result back the master.

Neptune[51] works as graph database service to work with highly connected datasets. Neptune stores RDF graph in the form of a quad containing four things Amazon S3. In addition to subject, predicate and object, graph identifier is also added. Neptune maintains three index permuations, SPOG, POGS and GPSO. In addition to three index permutations, each index permutations 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 [52]. Gremlin queries can be processed into a series of TinkerPop steps, which produce the accurate results, they are not much efficient when applied on large graphs. Neptune converts these steps into its custom query graph steps. In case of failed conversion, process is stopped. Neptune makes use of cardinality estimations and static analysis to rewrite queries, which will be executed through pipeline of physical operators. In Neptune there is no partitioning of graphs, instead it maintains 15 replicas at its master node.

The work in **Adaptive Distributed RDF Graph Fragmentation and Allocation based on Query Workload** [100] stores RDF data in three types of fragmentation. Vertical, Horizontal, and Mixed Fragmentation are used in the study. In comparison with other systems this study is listed as HF, VP and MF. Storage is performed on the basis of FAP (frequently accessed patterns) from the query workload. In the study to improve the query throughout VF is used, to maximize the parallelism of query evaluation and reduction of the query response time for a single query, HF is employed. The MF combines both the VF and HF. The study (HF, VF, MF) uses no indexing mechanism. SPARQL query processing in a study **(HF,VF, MF)** is based on the efficient fragmentation across different partitions. Due to implicit replications in three types of fragmentation strategies used, there will be different query decomposition for each fragmentation strategy. To avoid this problem, a cost model-driven approach is used in this study. Query optimization technique of pushing cross-fragments joins to avoid useless intermediate results.

DiStRDF [101] stores RDF data in an encoded form. DiStRDF manages the storage of Spatio-temporal RDF data through its storage layer. It employs a special purpose encoding scheme [115] for the encoding of RDF data stored in CSV or Parquet. It also maintains a dictionary for mapping between ID's and data. DiStRDF can handle data stored in both the Triple table and property table. Due to the huge size of RDF data, it exploits range partition to distribute data. For indexing, DiStRDF makes use of 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. For SPARQL query execution, DiStRDF Processing Layer comes into action. For the implementation of processing layer of DiStRDF, Spark, a distributed in-memory processing engine is used. Query processing involves different steps, i.e., the encoding of data is followed by different logical plans using physical plans. Query processing in DiStRDF is performed by storing data in TT and PT.

Leon [102] stores RDF data in heuristics-based partitioning, which is based on the characteristics set. Characteristics set is a mean to capture the structure of a dataset. Lean save the triples evenly on different partitions after encoding. Leon maintains 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. Leon effectively deals with the multi-query problem. By exploiting the characteristics set based partitioning, Leon minimizes the communication cost during query evaluation.

DISE [103] stores RDF data as tensors. RDF triples can be represented as a 3D tensor in which the adjacency matrix of subject, predicate and object is known as slice. According to [116] Tensors can be shown as a multidimensional array of ordered columns. DISE maintains indexes of RDD of subject, predicate and object at their respective position. DICE performs the translation of SPARQL queries into Spark tensor operation through Spark-Scala compliant code. DISE integrates the tensor representation of RDF into the representation layer and tensor querying into the query layer of SANSA stack. To execute a query on RDF as tensor data, DISE makes use of SANSA RDF reader. After input, the file is converted to RDD of triples. In the next step, RDDs of triples creates tensor. Next step creates the Apache Jena query objects from the input query string. At this step, the Degree of Freedom of each query pattern is calculated by recursively scanning. For reducing the search space, DICE starts with the selection of query pattern with lowest DOF. DOF of SPARQL query is "a measure of a triple pattern's explicit constraints" [117]. Based on the results of previous steps of translating SPARQL query into tensor operation, the builder calculates the results of the query.

10   SPARQL Benchmarks for RDF Engines

This discussion is adapted from [36] 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

---

[51]https://aws.amazon.com/neptune/
[52]https://dbdb.io/db/neptune

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

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 [118]. Real-world RDF datasets are very useful as they contain real world information [119]. 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 [120]. In order to select a datasets for benchmarking RDF engines, two measures are proposed by [36]. These two measures are (1) Dataset Structuredness, (2) Relationship Speciality. The *structuredness* or *coherence* measures how well a dataset's classes (i.e., `rdf:type`) are covered by the different instances of the dataset. The structuredness value for any given dataset lies between $[0, 1]$, where 0 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 number of occurrences of a predicate associated with each resource in the dataset provides useful information on the graph structure of an RDF dataset and makes some resources distinguishable from others [121]. In real datasets, this kind of *relationship speciality is commonplace*. The relationship speciality of RDF datasets can be any positive natural number: the higher the number, the higher the relationship speciality. The dataset structuredness and relationship speciality directly affect the result size, the number of intermediate results, and the selectivities of the triple patterns of the given SPARQL query [36]. The formal definitions of these datasets measures can be found in [36]. It is essential to mention that besides the dataset structuredness and relationship speciality, observations from the literature [24, 120] suggest that other RDF datasets feature such as varying number of triples, number of resources, number of properties, number of objects, number of classes, diversity in literal values, average properties and instances per class, average indegrees and outdegrees as well as their distribution across resources should also be considered which selecting datasets for SPARQL benchmark.

SPARQL Queries. Saleem et al. [36] suggests that a SPARQL querying benchmark should vary the queries with respect to various features such as *query characteristics*: number of projection variables, number of triple patterns, result set sizes, query execution time, number of BGPs, number of join vertices, mean join vertex degree, mean triple pattern selectivities, join vertex types, and highly used SPARQL clauses (e.g., `LIMIT`, `OPTIONAL`, `ORDER BY`, `DISTINCT`, `UNION`, `FILTER`, `REGEX`). All of these features have a direct impact on the runtime performance of RDF engines. Saleem et al. [36] combine all these vital query features into a single composite metric 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 be the mean and σ_i the standard deviation of a given distribution w.r.t. the ith feature of the said distribution. The overall diversity score DS of the queries is the average coefficient of variation of all the query features k 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. [36] divided the performance metrics for RDF engines comparison into four broader categories:

- **Query Processing Related**: The measures in this category are about query processing of the RDF engines, for which query execution time is the most important. However, since a benchmark usually contains many queries, reporting the execution time for individual queries might not be possible. For this problem, the combined results are presented in terms of measures like Query Mix per Hour (QMpH) and Queries per Second (QpS) [36]. In addition, the query processing overhead in terms of the CPU and memory usage is important to measure during the query executions [122]. This also includes the number of intermediate results, the number of disk/memory swaps, etc.

- **Data Storage Related:** In this category, the data loading time, the storage space acquired, and the index size are important performance metrics [36].

- **Result Set Related:** A fair comparison between two RDF engine is only possible if they produce exactly the same results. Therefore, the result set correctness and completeness are important metrics to be considered in the RDF engines evaluations [36].

- **Parallelism with/without Updates:** There are benchmarks execution frameworks such IGUANA [123] that can be used to measure the parallel query processing capabilities of the triplestores by simulating workloads from multiple querying agents with and without dataset updates.

We now present a broader analysis of the state-of-the-art existing RDF engines benchmarks across all of the above-mentioned dataset and query features as well as the performance metrics.

10.2   RDF Engines Benchmarks Analysis

Saleem et al. [36] considered the benchmarks according to the following inclusion criteria: (1) the benchmark target the query runtime performance evaluation of triplestores, (2) both RDF data and SPARQL queries of the benchmark 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.

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) [124] is about railway networks in increasing sizes, which serializes them in different formats, including RDF. *The Waterloo SPARQL Diversity Test Suite* (WatDiv) [125] provides a synthetic data generator that produces RDF data with a tunable structuredness value and a query generator. The queries are generated from different query templates. *SP2Bench* [122] mirrors vital characteristics (such as power-law distributions or Gaussian curves) of the data in the DBLP bibliographic database. *The Berlin SPARQL Benchmark* (BSBM) [126] uses query templates to generate any number of SPARQL queries for benchmarking, covering multiple use cases such as explore, update, and business intelligence. *Bowlogna* [127] models a real-world setting derived from the Bologna process and offered mostly analytic queries reflecting data-intensive user needs. The *LDBC Social Network Benchmark* (SNB) defines two workloads. First, the *Interactive* workload (SNB-INT) measures the evaluation of graph patterns in a localized scope (e.g., in the neighbourhood of a person), with the graph being continuously updated [128]. Second, the *Business Intelligence* workload (SNB-BI) focuses on queries that mix complex graph pattern matching with aggregations, touching on a significant portion of the graph [129], without any updates. Note that these two workloads are regarded as two separate RDF engine benchmarks based on the same dataset.

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* [118] is a cluster-based SPARQL benchmark generator, which is able to synthesize customizable benchmarks from the query logs of SPARQL endpoints. The *DBpedia SPARQL Benchmark* (DBPSB) [119] is another cluster-based approach that generates benchmark queries from DBpedia query logs, but employs different clustering techniques than FEASIBLE. The *FishMark* [130] dataset is obtained from FishBase[53] and provided in both RDF and SQL versions. The SPARQL queries were obtained from logs of the web-based FishBase application. *BioBench* [131] evaluates the performance of RDF RDF engines with biological datasets and queries from five different real-world RDF datasets[54], i.e., Cell, Allie, PDBJ, DDBJ, and UniProt. Due to the size of the datasets, we were only able to analyze the combined data and queries of the first three.

Basic Statistics. Table 11 shows high-level statistics of the selected datasets and queries of the benchmarks. For the synthetic benchmarks that include data generators, the datasets used in the evaluation of the original paper were chosen. For template-based query generators such as WatDiv, DBPSB, SNB, we one query per available template was selected. For FEASIBLE (a benchmark generation framework from query logs), 50 queries from DBpedia log was selected, to be comparable with a well-known WatDiv benchmark that includes 20 basic testing query templates, and 30 extensions for testing.[55]

---

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

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

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

	Benchmark	Subjects	Predicates	Objects	Triples	Queries
Synthetic	Bowlogna [127]	2,151k	39	260k	12M	16
	TrainB. [124]	3,355k	16	3,357k	41M	11
	BSBM [126]	9,039k	40	14,966k	100M	20
	SP2Bench [122]	7,002k	5,718	19,347k	49M	14
	WatDiv [125]	5,212k	86	9,753k	108M	50
	SNB [128, 129]	7,193k	40	17,544k	46M	21
Real	FishMark [130]	395k	878	1,148k	10M	22
	BioBench [131]	278,007k	299	232,041k	1,451M	39
	FEASIBLE [118]	18,425k	39,672	65,184k	232M	50
	DBPSB [119]	18,425k	39,672	65,184k	232M	25

Table 11: High-level statistics of the data and queries in the benchmarks. Both SNB-BI and SNB-INT use the same dataset and are therefore named as SNB for simplicity.

(a) Structuredness

(b) Relationship Specialty

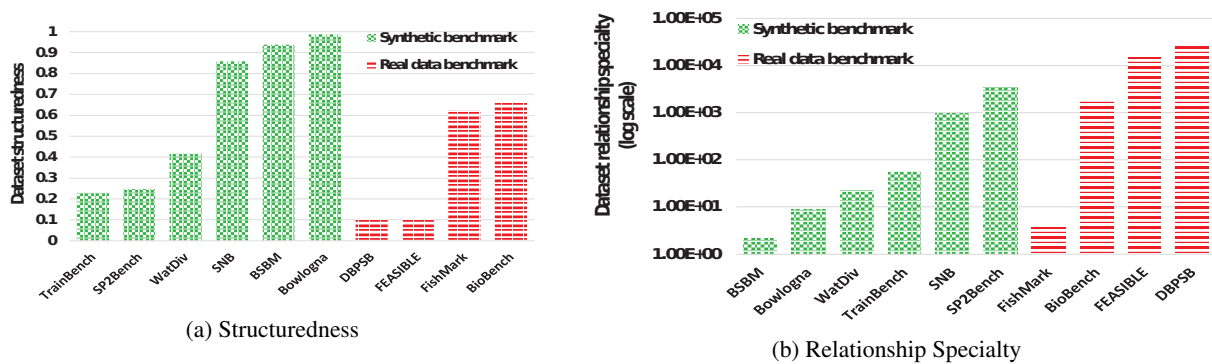



Figure 13: Analysis of the datasets of the RDF engines benchmarks.

Structuredness and Relationship Speciality. Figure 13a contains the structuredness values of the different benchmarks. Duan et al. [120] 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 [121], it is evident that the values of relationship speciality shown by real world 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.

Table 12 shows each benchmark with its coverage percentage of SPARQL clauses and join vertexes [37]. The table also highlights the benchmarks with missing SPARQL constructs and types of overused join vertexes. 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[118] and WatDiv[16], 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.

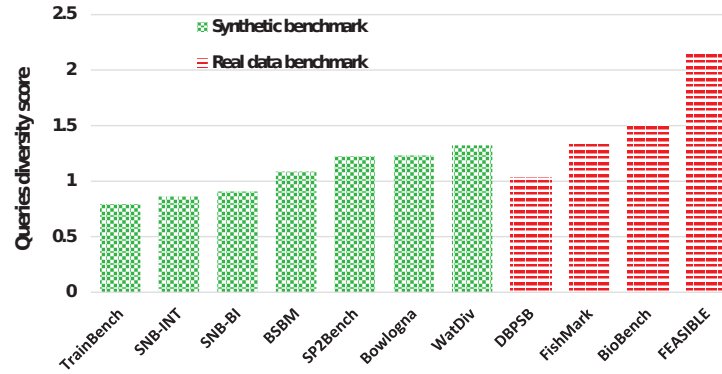



Figure 14: Benchmarks with diversity counts

	Benchmark	SPARQL Clauses							Types of Join Vertexes				
		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 12: Percentages of different SPARQL clauses and types of join vertexes 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.

	Benchmark	Processing			Storage			Result Set		Additional	
		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	✗	✗	✗	✓	✓	✗	✓	✗	✓	✗

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.

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 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.

References

[1]   Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. Rdf data storage and query processing schemes: A survey. *ACM Comput. Surv.*, 51(4):84:1–84:36, September 2018.

[2]   Zhengyu Pan, Tao Zhu, Hong Liu, and Huansheng Ning. A survey of RDF management technologies and benchmark datasets. *Journal of Ambient Intelligence and Humanized Computing*, 9(5):1693–1704, oct 2018.

[3]   Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.

[4]   Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos 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.

[5]   Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed Mehdi Reza Beheshti, and Sherif Sakr. DREAM: Distributed RDF engine with adaptive query planner and minimal communication. *Proceedings of the VLDB Endowment*, 8(6):654–665, feb 2015.

[6]   Luis Galárraga, Katja Hose, and Ralf 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.

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

[8]   Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. Processing sparql queries over distributed rdf graphs. *The VLDB Journal*, 25(2):243–268, April 2016.

[9]   Alexander Schätzle, Martin Przyjaciel-Zablocki, Antony Neu, and Georg 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.

[10]   Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF querying with SPARQL on Spark. *Proceedings of the VLDB Endowment*, 9(10):804–815, 2016.

[11]   Adnan Akhter, Axel-Cyrille Ngomo Ngonga, and Muhammad Saleem. An empirical evaluation of rdf graph partitioning techniques. In *European Knowledge Acquisition Workshop*, pages 3–18. Springer, 2018.

[12] James Groff and Paul Weinberg. *SQL The Complete Reference, 3rd Edition*. McGraw-Hill, Inc., USA, 3 edition, 2009.

[13] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew 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.

[14] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. Feasible: A feature-based sparql benchmark generation framework. In *International Semantic Web Conference*, pages 52–69. Springer, 2015.

[15] Felix Conrads, Jens Lehmann, Muhammad Saleem, Mohamed Morsey, and Axel-Cyrille Ngonga 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.

[16] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of rdf data management systems. In *International Semantic Web Conference*, pages 197–212. Springer, 2014.

[17] David Célestin Faye, Olivier Curé, and Guillaume 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.

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

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

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

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

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

[23] Muhammad Saleem, Alexander Potocki, Tommaso Soru, Olaf Hartig, and Axel-Cyrille Ngonga Ngomo. CostFed: Cost-based query optimization for SPARQL endpoint federation. In *SEMANTICS*, volume 137 of *Procedia Computer Science*, pages 163–174. Elsevier, 2018.

[24] Muhammad Saleem, Ali Hasnain, and Axel-Cyrille Ngonga Ngomo. LargeRDFBench: A billion triples benchmark for SPARQL endpoint federation. *J. Web Sem.*, 48:85–125, 2018.

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

[26] Sherif Sakr, Marcin Wylot, Raghava Mutharaju, Danh Le Phuoc, and Irini Fundulaki. *Linked Data: Storing, Querying, and Reasoning*. Springer, 2018.

[27] Nahla Mohammed Elzein, Mazlina Abdul Majid, Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Fadele Ayotunde Alaba, and Muhammad Imran. Managing big rdf data in clouds: Challenges, opportunities, and solutions. *Sustainable Cities and Society*, 39:375 – 386, 2018.

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

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

[30] Muhammad Qasim Yasin, Xiaowang Zhang, Rafiul Haq, Zhiyong Feng, and Sofonias 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.

[31] Khadija 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.

[32] Luiz Henrique Zambom Santana and Ronaldo dos Santos 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.

[33] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave 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.

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

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

[36] Muhammad Saleem, Gábor Szárnyas, Felix Conrads, Syed Ahmad Chan Bukhari, Qaiser Mehmood, and Axel-Cyrille 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.

[37] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. LSQ: the linked SPARQL queries dataset. In *ISWC*, pages 261–269. Springer, 2015.

[38] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate 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.

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

[40] Matthias Bröcheler, Andrea Pugliese, and V. S. Subrahmanian. Dogma: A disk-oriented graph matching algorithm for rdf databases. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009*, pages 97–113, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[41] Vijay Ingalalli, Dino Ienco, and Pascal Poncelet. *Querying RDF Data: a Multigraph-based Approach*, chapter 5, pages 135–165. John Wiley Sons, Ltd, 2018.

[42] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min 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.

[43] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios 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.

[44] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In Ian Horrocks and James 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.

[45] David 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.

[46] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. Triplebit: A fast and compact system for large scale rdf data. *Proc. VLDB Endow.*, 6(7):517–528, May 2013.

[47] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan 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.

[48] Dave Kolas, Ian Emmons, and Mike Dean. Efficient linked-list RDF indexing in Parliament. *CEUR Workshop Proceedings*, 517:17–32, 2009.

[49] Claus Stadler, Gezim Sejdiu, Damien Graux, and Jens Lehmann. Sparklify: A scalable software component for efficient evaluation of sparql queries over distributed rdf datasets. In Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtěch Svátek, Isabel Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon, editors, *The Semantic Web – ISWC 2019*, pages 293–308, Cham, 2019. Springer International Publishing.

[50] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering sparql queries over relational databases. *Semantic Web*, 8(3):471–487, 2017.

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

[52] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[53] Muhammad Saleem, Yasar Khan, Ali Hasnain, Ivan Ermilov, and Axel-Cyrille Ngonga Ngomo. A fine-grained evaluation of sparql endpoint federation systems. *Semantic Web*, 7(5):493–518, 2016.

[54] Daniel Janke, Steffen Staab, and Matthias 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.

[55] Aydin Buluc, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning, 2013.

[56] Stephen Harris and Nicholas 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.

[57] Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao 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.

[58] Maciej Janik and Krys 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.

[59] Andreas Harth and Stefan 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.

[60] David Wood, Paul Gearon, and Tom Adams. Kowari: A platform for semantic web storage and analysis. In *XTech 2005 Conference*, pages 5–402, 2005.

[61] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan 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.

[62] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[63] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. Taming subgraph isomorphism for rdf query processing. *Proceedings of the VLDB Endowment*, 8(11), 2015.

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

[65] Anas Katib, Vasil Slavov, and Praveen Rao. Riq: Fast processing of sparql queries on rdf quadruples. *Journal of Web Semantics*, 37:90–111, 2016.

[66] Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. Strabon: A semantic geospatial dbms. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web – ISWC 2012*, pages 295–311, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[67] Medha Atre and James 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.

[68] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan 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.

[69] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. Rdfox: A highly-scalable rdf store. In Marcelo Arenas, Oscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015*, pages 3–20, Cham, 2015. Springer International Publishing.

[70] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan 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.

[71] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[72] Uwe 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.

[73] Manolis Koubarakis and Kostis 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.

[74] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan 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.

[75] Kurt Rohloff and Richard 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.

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

[77] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. An intermediate algebra for optimizing rdf graph pattern matching on mapreduce. In Grigoris Antoniou, Marko Grobelnik, Elena Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Pan, editors, *The Semantic Web: Research and Applications*, pages 46–61, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[78] Vaibhav Khadilkar, Murat Kantarcioglu, Bhavani Thuraisingham, and Paolo 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.

[79] Roshan Punnoose, Adina Crainiceanu, and David 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.

[80] Andrés Aranda-Andújar, Francesca Bugiotti, Jesús Camacho-Rodríguez, Dario Colazzo, François Goasdoué, Zoi Kaoudi, and Ioana 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.

[81] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios 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.

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

[83] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *Proceedings of the International Workshop on Semantic Web Information Management, SWIM 2011*, 2011.

[84] Raghava Mutharaju, Sherif Sakr, Alessandra Sala, and Pascal 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.

[85] 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.

[86] Philip Stutz, Mihaela Verman, Lorenz Fischer, and Abraham 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.

[87] Günes Aluç, M. Tamer Özsu, Khuzaima Daudjee, and Olaf Hartig. chameleon-db: a workload-aware robust rdf data management system. 2013.

[88] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin 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.

[89] Xi Chen, Huajun Chen, Ningyu Zhang, and Songyang 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.

[90] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Hai Jin, and Ling 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.

[91] 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.

[92] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB Journal*, 25(3):355–380, jun 2016.

[93] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thorsten Berberich, and Georg Lausen. In Fusheng Wang, Gang Luo, Chunhua Weng, Arijit Khan, Prasenjit Mitra, and Cong 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.

[94] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei 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.

[95] Jens Lehmann, Gezim Sejdiu, Lorenz Bühmann, Patrick Westphal, Claus Stadler, Ivan Ermilov, Simon Bin, Nilesh Chakraborty, Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, and Hajira Jabeen. Distributed semantic analytics using the sansa stack. In Claudia d'Amato, Miriam Fernandez, Valentina Tamma, Freddy Lecue, Philippe Cudré-Mauroux, Juan Sequeda, Christoph Lange, and Jeff Heflin, editors, *The Semantic Web – ISWC 2017*, pages 147–155, Cham, 2017. Springer International Publishing.

[96] Liang He, Bin Shao, Yatao Li, Huanhuan Xia, Yanghua Xiao, Enhong Chen, and Liang Jeff Chen. Stylus: A strongly-typed store for serving massive rdf data. *Proc. VLDB Endow.*, 11(2):203–216, October 2017.

[97] Daniel Janke, Steffen Staab, and Matthias Thimm. Koral: A glass box profiling system for individual components of distributed rdf stores. In *BLINK/NLIWoD3@ISWC*, 2017.

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

[99] Amgad Madkour, Ahmed M. Aly, and Walid G. Aref. Worq: Workload-driven rdf query processing. In Denny Vrandečić, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, editors, *The Semantic Web – ISWC 2018*, pages 583–599, Cham, 2018. Springer International Publishing.

[100] 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.

[101] Randall T. Whitman, Bryan G. Marsh, Michael B. Park, and Erik G. Hoel. Distributed spatial and spatio-temporal join on apache spark. *ACM Trans. Spatial Algorithms Syst.*, 5(1), June 2019.

[102] Xintong Guo, Hong Gao, and Zhaonian Zou. Leon: A distributed rdf engine for multi-query processing. In Guoliang Li, Jun Yang, Joao Gama, Juggapong Natwichai, and Yongxin Tong, editors, *Database Systems for Advanced Applications*, pages 742–759, Cham, 2019. Springer International Publishing.

[103] 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.

[104] Kevin Wilkinson. Jena property table implementation, 2006.

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

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

[107] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: Graph algorithms for the (semantic) web. In Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm, editors, *The Semantic Web – ISWC 2010*, pages 764–780, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[108] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR 2015 - 7th Biennial Conference on Innovative Data Systems Research*, 2015.

[109] Marcin Wylot, Jigé Pont, Mariusz Wisniewski, and Philippe 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.

[110] Damien Graux, Louis Jachiet, Pierre Genevès, and Nabil Layaïda. SPARQLGX: Efficient distributed evaluation of SPARQL with apache spark. In Paul Groth, Elena Simperl, Alasdair Gray, Marta Sabou, Markus Krötzsch, Freddy Lecue, Fabian Flöck, and Yolanda 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.

[111] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert 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.

[112] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion 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.

[113] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion 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.

[114] Peng Peng, Lei Zou, and Runyu Guan. Accelerating partial evaluation in distributed SPARQL query evaluation. *CoRR*, abs/1902.03700, 2019.

[115] Akrivi Vlachou, Christos Doulkeridis, Apostolos Glenis, Georgios M. Santipantakis, and George 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.

[116] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM REVIEW*, 51(3):455–500, 2009.

[117] Roberto De Virgilio. Distributed in-memory sparql processing via dof analysis. In *EDBT*, 2017.

[118] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. FEASIBLE: a feature-based SPARQL benchmark generation framework. In *ISWC*, pages 52–69. Springer, 2015.

[119] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. DBpedia SPARQL benchmark - performance assessment with real queries on real data. In *ISWC*, pages 454–469. Springer, 2011.

[120] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: A comparison of RDF benchmarks and real RDF datasets. In *SIGMOD*, pages 145–156. ACM, 2011.

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

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

[123] Felix Conrads, Jens Lehmann, Muhammad Saleem, Mohamed Morsey, and Axel-Cyrille Ngonga Ngomo. IGUANA: A generic framework for benchmarking the read-write performance of triple stores. In *ISWC*, pages 48–65. Springer, 2017.

[124] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. The Train Benchmark: Cross-technology performance evaluation of continuous model queries. *Softw. Syst. Model.*, 17(4):1365–1393, 2018.

[125] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212. 2014.

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

[127] Gianluca Demartini, Iliya Enchev, Marcin Wylot, Joël Gapany, and Philippe Cudré-Mauroux. BowlognaBench - benchmarking RDF analytics. In *Data-Driven Process Discovery and Analysis SIMPDA*, pages 82–102. Springer, 2011.

[128] Orri Erling, Alex Averbuch, Josep-Lluis Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. The LDBC Social Network Benchmark: Interactive workload. In *SIGMOD*, pages 619–630. ACM, 2015.

[129] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter A. Boncz, Vlad Haprian, and János Benjamin 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.

[130] Samantha Bail, Sandra Alkiviadous, Bijan Parsia, David Workman, Mark van Harmelen, Rafael S. Gonçalves, and Cristina 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.

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