A Survey of RDF Stores & SPARQL Engines
for Querying Knowledge Graphs

**Waqas Ali · Muhammad Saleem · Bin Yao ·
Aidan Hogan · Axel-Cyrille Ngonga Ngomo**

Abstract RDF has seen increased adoption in recent years,
prompting the standardization of the SPARQL query lan-
guage for RDF, and the development of local and distributed
engines for processing SPARQL queries. This survey paper
provides a comprehensive review of techniques and systems
for querying RDF knowledge graphs. While other reviews
on this topic tend to focus on the distributed setting, the
main focus of the work is on providing a comprehensive sur-
vey of state-of-the-art storage, indexing and query process-
ing techniques for efficiently evaluating SPARQL queries in
a local setting (on one machine). To keep the survey self-
contained, we also provide a short discussion on graph par-
titioning techniques used in the distributed setting. We con-
clude by discussing contemporary research challenges for
further improving SPARQL query engines. This extended
version also provides a survey of over one hundred SPARQL
query engines and the techniques they use, along with twelve
benchmarks and their features.

Keywords Knowledge Graph · Storage · Indexing · Query
Processing · SPARQL

W. Ali
SEIEE, Shanghai Jiao Tong University, Shanghai, China.
E-mail: waqasali@sjtu.edu.cn

M. Saleem
AKSW, University of Leipzig, Leipzig, Germany.
E-mail: saleem@informatik.uni-leipzig.de

✉ B. Yao (Hangzhou Qianjiang Distinguished Expert)
SEIEE, Shanghai Jiao Tong University, Shanghai, China.
Hangzhou Institute of Advanced Technology, Hangzhou, China.
E-mail: yaobin@cs.sjtu.edu.cn

A. Hogan
DCC, University of Chile & IMFD, Santiago, Chile.
E-mail: ahogan@dcc.uchile.cl

A.-C. Ngonga Ngomo
DICE, Paderborn University, Paderborn, Germany.
E-mail: axel.ngonga@upb.de

1 Introduction

The Resource Description Framework (RDF) is a graph-
based data model where triples of the form (s, p, o) denote
directed labeled edges $s \xrightarrow{p} o$ in a graph. RDF has gained
significant adoption in the past years, particularly on the
Web. As of 2019, over 5 million websites publish RDF data
embedded in their webpages [34]. RDF has also become
a popular format for publishing knowledge graphs on the
Web, the largest of which – including Bio2RDF, DBpedia,
PubChemRDF, UniProt, and Wikidata – contain billions of
triples. These developments have brought about the need for
optimized techniques and engines for querying large RDF
graphs. We refer to engines that allow for storing, indexing
and processing joins over RDF as *RDF stores*.

While various query languages have historically been
proposed for RDF, the SPARQL Protocol and RDF Query
Language (SPARQL) has become the standard [92]. The
first version of SPARQL was standardized in 2008, while
SPARQL 1.1 was released in 2013 [92]. SPARQL is an ex-
pressive language that supports not only joins, but also vari-
ants of the broader relational algebra (projection, selection,
union, difference, etc.). Various new features were added in
SPARQL 1.1, such as *property paths* for matching arbitrary-
length paths in the RDF graph. Hundreds of SPARQL query
services, called *endpoints*, have emerged on the Web [43],
with the most popular endpoints receiving millions of queries
per day [197, 148]. We refer to engines that support storing,
indexing and processing SPARQL (1.1) queries over RDF
as *SPARQL engines*. Since SPARQL supports joins, we con-
sider any SPARQL engine to also be an RDF store.

Efficient data storage, indexing and join processing are
key to RDF stores (and thus, to SPARQL engines):

- *Storage.* Different engines store RDF data using differ-
  ent structures (tables, graphs, etc.), encodings (integer

IDs, string compression, etc.) and media (main memory, disk, etc.). Which storage to use may depend on the scale of the data, the types of query features supported, etc.

- *Indexing.* Indexes are used in RDF stores for fast lookups and query execution. Different index types can support different operations with varying time–space trade-offs.
- *Join Processing.* At the core of evaluating queries lie efficient methods for processing joins. Aside from traditional pairwise joins, recent years have seen the emergence of novel techniques, such as multiway and worst-case optimal joins, as well as GPU-based join processing. Optimizing the order of evaluation of joins can also be important to ensure efficient processing.

Beyond processing joins, SPARQL engines must offer efficient support for more expressive query features:

- *Query Processing.* SPARQL is an expressive language containing a variety of query features beyond joins that need to be supported efficiently, such as filter expressions, optionals, path queries, etc.

RDF stores can further be divided into two categories: (1) *local stores* (also called single-node stores) that manage RDF data on one machine and (2) *distributed stores* that partition RDF data over multiple machines. While local stores are more lightweight, the resources of one machine limit scalability [249, 175, 104]. Various kinds of distributed RDF stores have thus been proposed [88, 104, 203, 204] that typically run on clusters of shared-nothing machines.

In this survey, we describe storage, indexing, join processing and query processing techniques employed by local RDF stores, as well as high-level strategies for partitioning RDF graphs as needed for distributed storage. An appendix in this extended version further compares 135 local and distributed RDF engines in terms of the techniques they use, as well as 12 benchmarks in terms of the types of data and queries they contain. The goal of this survey is to give a succinct introduction of the different techniques used by RDF query engines, and also to help users to choose the appropriate engine or benchmark for a given use case.

The rest of the paper is structured as follows. Section 2 discusses and contrasts this survey with related literature. Section 3 provides preliminaries for RDF and SPARQL. Sections 4, 5, 6 and 7 review techniques for storage, indexing, join processing and query processing, respectively. Section 8 explains different graph partitioning techniques for distributing storage over multiple machines. Section 9 introduces additional content available in the appendix of this extended version, which surveys 135 local and distributed RDF engines, along with 12 SPARQL benchmarks. Section 10 concludes the paper with subsections for current trends and research challenges regarding efficient RDF-based data management and query processing.

2 Literature Review

We first discuss related studies. More specifically, we summarize peer-reviewed tertiary literature (surveys in journals, short surveys in proceedings, book chapters, surveys with empirical comparisons, etc.) from the last 10 years collating techniques, engines and/or benchmarks for querying RDF. We summarize the topics covered by these works in Table 1. We use ✓, ∼ and blank cells to denote detailed, partial or little/no discussion, respectively, when compared with the current survey (the bottom row). We also present the number of engines and benchmarks included in the extended version of this survey. If the respective publication does not formally list all systems/benchmarks (e.g., as a table), we may write $n+$ as an estimate for the number discussed in the text.

Sakr et al. [196] present three schemes for storing RDF data in relational databases, surveying works that use the different schemes. Svoboda et al. [221] provide a brief survey on indexing schemes for RDF divided into three categories: local, distributed and global. Faye et al. [70] focus on both storage and indexing schemes for local RDF engines, divided into native and non-native storage schemes. Luo et al. [141] also focus on RDF storage and indexing schemes under the relational-, entity-, and graph-based perspectives in local RDF engines. Compared to these works, we present join processing, query processing and partitioning techniques; furthermore, these works predate the standardization of SPARQL 1.1, and thus our discussion includes more recent storage and indexing techniques, as well as support for new features such as property paths.

Later surveys began to focus on distributed RDF stores. Kaoudi et al. [116] present a survey of RDF stores explicitly designed for a cloud-based environment. Ma et al. [144] provide an overview of RDF storage in relational and NoSQL databases. Özsu [174] presents a survey that focuses on storage techniques for RDF within local and distributed stores, with a brief overview of query processing techniques in distributed and decentralized (Linked Data) settings. Abdelaziz et al. [3] survey 22 distributed RDF stores, and compare 12 experimentally in terms of pre-processing cost, query performance, scalability, and workload adaptability. Elzein et al. [67] present a survey on the storage and query processing techniques used by RDF stores on the cloud. Janke & Staab [111] present lecture notes discussing RDF graph partitioning, indexing, and query processing techniques, with a focus on distributed and cloud-based RDF engines. Pan et al. [175] provide an overview of local and distributed storage schemes for RDF. Yasin et al. [255] discussed SPARQL (1.1) query processing in the context of distributed RDF stores. Wylot et al. [249] present a comprehensive survey of storage and indexing techniques for local (centralized) and distributed RDF stores, along with a discussion of benchmarks; most of their survey is dedicated to distributed and

Table 1: Prior tertiary literature on RDF query engines; the abbreviations are: **Sto./**Storage, **Ind./**Indexing, **J.Pr./**Join Processing, **Q.Pr./**Query Processing, **Dis./**Distribution, **Eng./**Engines, **Ben./**Benchmarks

| Study | Year | Techniques | | | | | Eng. | Bench. |
		Sto.	Ind.	J.Pr.	Q.Pr.	Dis.		
Sakr et al. [196]	2010	✓	∼				10+	
Svoboda et al. [221]	2011	∼	∼			∼	14	6
Faye et al. [70]	2012	✓	∼	∼			13	
Luo et al. [141]	2012	✓	✓				20+	
Kaoudi et al. [116]	2015	✓	∼	∼	∼	✓	17	
Ma et al. [144]	2016	✓	∼			∼	17	6
Özsu [174]	2016	✓		∼	∼	∼	35+	
Abdelaziz et al. [3]	2017	✓	∼	∼	∼	✓	21	4
Elzein et al. [67]	2018	✓		∼	∼	∼	15+	
Janke & Staab [111]	2018	∼	∼	∼	∼	✓	50+	9
Pan et al. [175]	2018	✓	∼	∼		∼	25+	4
Wylot et. al [249]	2018	✓	∼	∼	∼	✓	24	8
Yasin et al. [255]	2018			∼	✓	∼	14	
Alaoui [9]	2019	✓				∼	30+	
Chawla et al. [50]	2020	✓	∼	∼		∼	46	9
Zambom et al. [257]	2020	∼			∼	✓	24	
Ali et al.		✓	✓	✓	✓	✓	135	12

federated stores. Alaoui [9] proposes a categorization scheme for RDF engines, including memory-, cloud-, graph- and binary-bases stores. The survey by Chawla et al. [50] reviews distributed RDF engines in terms of storage, partitioning, indexing, and retrieval. The short survey by Zambom & dos Santos [257] discusses mapping RDF data into NoSQL databases. All of these works focus on techniques for storing RDF, particularly in distributed settings, where our survey is more detailed in terms of join and query processing techniques, particularly in local settings.

Local RDF stores are those most commonly found in practice [43]. To the best of our knowledge, our survey provides the most comprehensive discussion thus far on storage, indexing, join processing and querying processing techniques for SPARQL in a local setting, where, for example, we discuss novel techniques for established features – such as novel indexing techniques based on compact data structures, worst-case optimal and matrix-based join processing techniques, multi-query optimization, etc. – as well as techniques for novel features in SPARQL 1.1 – such as indexing and query processing techniques for evaluating property paths – that are not well-represented in the existing literature. To keep our survey self-contained, we also present partitioning techniques for RDF graphs, and include distributed stores and benchmarks in our survey. Per Table 1, the survey of engines and benchmarks found in the online version is more comprehensive than seen in previous works [10]. Conversely, some of the aforementioned works are more detailed in certain aspects, particularly distributed stores; we refer to this literature for further details as appropriate.

3 Preliminaries

Before beginning the core of the survey, we first introduce some preliminaries regarding RDF and SPARQL.

3.1 RDF

The RDF data model [208] uses *RDF terms* from three pairwise disjoint sets: the set **I** of *Internationalized Resource Identifiers* (*IRIs*) [66] used to identify resources; the set **L** of *literals* used for (language-tagged or plain) strings and datatype values; and the set **B** of *blank nodes*, interpreted as existential variables. An *RDF triple* $(s, p, o) \in \mathbf{IB} \times \mathbf{I} \times \mathbf{IBL}$ contains a subject s, a predicate p and an object o.[1] A set of RDF terms is called an *RDF graph* G, where each triple $(s, p, o) \in G$ represents a directed labeled edge $s \xrightarrow{p} o$. The sets $\mathrm{s}(G)$, $\mathrm{p}(G)$ and $\mathrm{o}(G)$ stand for the set of subjects, predicates and objects in G, respectively. We further denote the set of *nodes* in G by $\mathrm{so}(G) := \mathrm{s}(G) \cup \mathrm{o}(G)$.

An example RDF graph, representing information about two university students, is shown in Figure 1. We include both a graphical representation and a triple-based representation. RDF terms such as `:DB`, `foaf:age`, etc., denote prefixed IRIs.[2] For example, `foaf:age` stands for the full IRI `http://xmlns.com/foaf/0.1/age` if we define the prefix `foaf` as `http://xmlns.com/foaf/0.1/`. Terms such as `"Motor RDF"@es` denote strings with (optional) language

---

[1] In this paper, we abbreviate the union of sets $M_1 \cup \ldots \cup M_n$ with $M_1 \ldots M_n$. Hence, **IBL** stands for $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$.

[2] We use the blank prefix (e.g., `:DB`) as an arbitrary example. Other prefixes used can be retrieved at `http://prefix.cc/`.

tags, and terms such as `"21"^^xsd:int` denote datatype values. Finally we denote blank nodes with the underscore prefix, where `_:p` refers to the existence of a project shared by Alice and Bob. Terms used in the predicate position (e.g., `foaf:age`, `skos:broader`) are known as *properties*. RDF defines the special property `rdf:type`, which indicates the *class* (e.g., `foaf:Person`, `foaf:Project`) of a resource.

The semantics of RDF can be defined using RDF Schema (RDFS) [37], covering class and property hierarchies, property domains and ranges, etc. Further semantics can be captured with the Web Ontology Language (OWL) [97], such as class and property equivalence; inverse, transitive, symmetric and reflexive properties; set- and restriction-based class definitions; and more besides. Since our focus is on querying RDF graphs, we do not discuss these standards in detail.

3.2 SPARQL

Various query languages for RDF have been proposed down through the years, such as RQL [118], SeRQL [218], etc. We focus our discussion on SPARQL [92], which is now the standard language for querying RDF, and refer to the work by Haase et al. [87] for information on its predecessors.

We define the core of SPARQL in terms of *basic graph patterns* that express the core pattern matched against an RDF graph; *navigational graph patterns* that match arbitrary-length paths; *complex graph patterns* that introduce various language features, such as OPTIONAL, UNION, MINUS, etc. [16]; and *query types* that specify what result to return.

Basic Graph Patterns (BGPs) At the core of SPARQL lie *triple patterns*, which are RDF triples that allow variables from the set $\mathbf{V}$ (disjoint with $\mathbf{IBL}$) in any position. A *basic graph pattern* (*BGP*) is a set of triple patterns. Since blank nodes in BGPs act as variables, we assume they have been replaced with variables. We use $\mathrm{vars}(B)$ to denote the set of variables in the BGP B. Given an RDF graph G, the *evaluation* of a BGP B, denoted $B(G)$, returns a set of *solution mappings*. A solution mapping μ is a partial mapping from the set $\mathbf{V}$ of variables to the set of RDF terms $\mathbf{IBL}$. We write $\mathrm{dm}(\mu)$ to denote the set of variables for which μ is defined. Given a triple pattern t, we use $\mu(t)$ to refer to the image of t under μ, i.e., the result of replacing any variable $v \in \mathrm{dm}(\mu)$ appearing in t with $\mu(v)$. $\mu(B)$ stands for the image of the BGP B under μ; i.e., $\mu(B) := \{\mu(t) \mid t \in B\}$. The evaluation of a BGP B on an RDF graph G is then given as $B(G) := \{\mu \mid \mu(B) \subseteq G \text{ and } \mathrm{dm}(\mu) = \mathrm{vars}(B)\}$. In the case of a singleton BGP $\{t\}$, we may write $\{t\}(G)$ as $t(G)$.

In Figure 2, we provide an example of a BGP along with its evaluation. Each row of the results refers to a solution mapping. Some solutions map different variables to the same term; each such solution is thus a homomorphism from the BGP to the RDF graph.

Table 2: Evaluation of path expressions

$p(G) := \{(s,o) \mid (s,p,o) \in G\}$
$\hat{}e(G) := \{(s,o) \mid (o,s) \in e(G)\}$
$e\texttt{+}(G) := \{(y_1, y_n) \mid \text{ for } 1 \le i < n : \exists(y_i, y_{i+1}) \in e(G)\}$
$e?(G) := e(G) \cup \{(x,x) \mid x \in \mathrm{so}(G)\}$
$e\texttt{*}(G) := (e\texttt{+})?(G)$
$e_1/e_2(G) := \{(x,z) \mid \exists y : (x,y) \in e_1(G) \land (y,z) \in e_2(G)\}$
$e_1\mid e_2(G) := e_1(G) \cup e_2(G)$
$!P(G) := \{(s,o) \mid (s,p,o) \in G \land p \notin P\}$
$!\hat{}P(G) := \{(s,o) \mid (o,p,s) \in G \land p \notin P\}$

Navigational Graph Patterns (NGPs) A key feature of graph query languages is the ability to match paths of arbitrary length [16]. In SPARQL (1.1), this ability is captured by *property paths* [92], which are regular expressions $\mathbf{E}$ that paths should match, defined recursively as follows:

– if p is an IRI, then p is a path expression (*property*);
– if e is a path expression, then $\hat{}e$ (*inverse*), $e\texttt{*}$ (*zero-or-more*, aka. *Kleene star*), $e\texttt{+}$ (*one-or-more*), and $e?$ (*zero-or-one*) are path expressions.
– If e_1, e_2 are path expressions, then e_1/e_2 (*concatenation*) and $e_1\mid e_2$ (*disjunction*) are path expressions.
– if P is a set of IRIs, then $!P$ and $!\hat{}P$ are path expressions (*negated property set*);[3]

The evaluation of path expressions on an RDF graph G returns pairs of nodes in G connected by paths that match the expression, as defined in Table 2. These path expressions are akin to *2-way regular path queries* (2RPQs) extended with negated property sets [128, 16].

We call a triple pattern (s, e, o) that further allows a path expression as the predicate (i.e., $e \in \mathbf{EV}$) a *path pattern*. A *navigational graph pattern* (NGP) is then a set of path patterns. Given a navigational graph pattern N, let $\mathrm{paths}(N) := \mathrm{p}(N) \cap \mathbf{E}$ denote the set of path expressions used in N. Given an RDF graph G and a set of path expressions $E \subseteq \mathbf{E}$, we denote by $G_E := G \cup (\bigcup_{e \in E}\{(s,e,o) \mid (s,o) \in e(G)\})$ the result of materializing all paths matching E in G. The evaluation of the navigational graph pattern N on G is then $N(G) := \{\mu \mid \mu(N) \subseteq G_{\mathrm{paths}(N)} \text{ and } \mathrm{dm}(\mu) = \mathrm{vars}(N)\}$.

We provide an example of a navigational graph pattern and its evaluation in Figure 3.

Complex Graph Patterns (CGPs) Complex graph patterns (CGPs) introduce additional language features that can combine and transform the results of one or more graph patterns. More specifically, evaluating BGPs and NGPs returns solution mappings that can be viewed as relations, (i.e., tables),

---

[3] SPARQL uses the syntax $!(p_1\mid\ldots\mid p_k\mid p_{k+1}\mid\ldots\mid p_n)$ which can be written as $!P\mid!\hat{}P'$, where $P = \{p_1, \ldots, p_k\}$ and $P' = \{p_{k+1}, \ldots, p_n\}$ [92, 128].

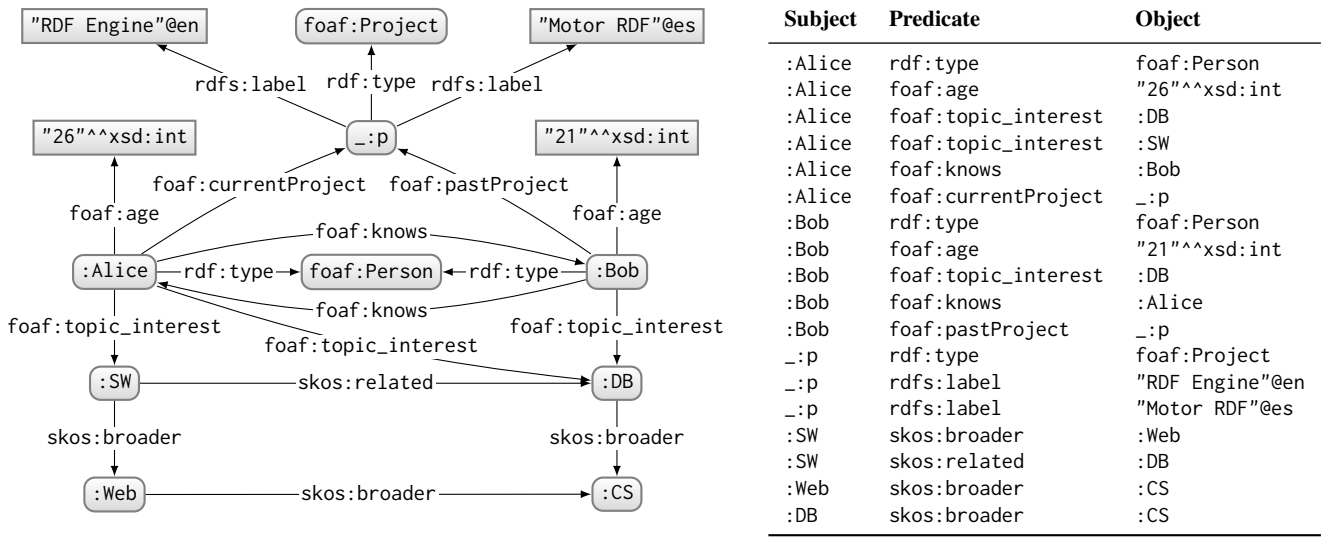| Subject	Predicate	Object
:Alice	rdf:type	foaf:Person
:Alice	foaf:age	"26"^^xsd:int
:Alice	foaf:topic_interest	:DB
:Alice	foaf:topic_interest	:SW
:Alice	foaf:knows	:Bob
:Alice	foaf:currentProject	_:p
:Bob	rdf:type	foaf:Person
:Bob	foaf:age	"21"^^xsd:int
:Bob	foaf:topic_interest	:DB
:Bob	foaf:knows	:Alice
:Bob	foaf:pastProject	_:p
_:p	rdf:type	foaf:Project
_:p	rdfs:label	"RDF Engine"@en
_:p	rdfs:label	"Motor RDF"@es
:SW	skos:broader	:Web
:SW	skos:related	:DB
:Web	skos:broader	:CS
:DB	skos:broader	:CS

Fig. 1: Graphical (left) and triple-based representation (right) of an example RDF graph

```
SELECT * WHERE {
  ?a a foaf:Person ; foaf:knows ?b ; foaf:topic_interest ?ia .
  ?b a foaf:Person ; foaf:knows ?a ; foaf:topic_interest ?ib .
}
```


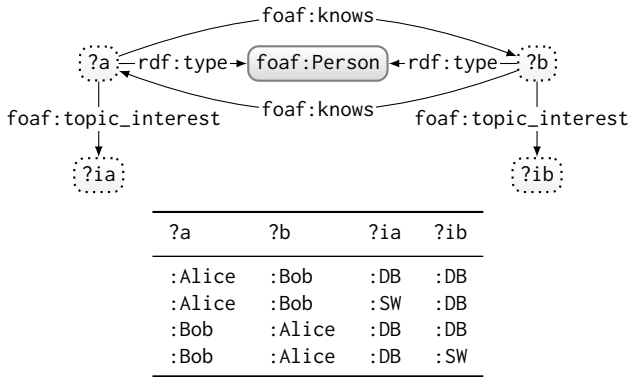| ?a	?b	?ia	?ib
:Alice	:Bob	:DB	:DB
:Alice	:Bob	:SW	:DB
:Bob	:Alice	:DB	:DB
:Bob	:Alice	:DB	:SW

Fig. 2: A BGP in SPARQL syntax and as a graph (above), with its evaluation over the graph of Figure 1 (below)

```
SELECT * WHERE {
  ?a a foaf:Person ; foaf:knows ?b ;
    foaf:topic_interest/skos:related*/foaf:topic_interest ?b .
  ?b a foaf:Person ; foaf:knows ?a .
}
```


foaf:topic_interest/skos:related*/^foaf:topic_interest

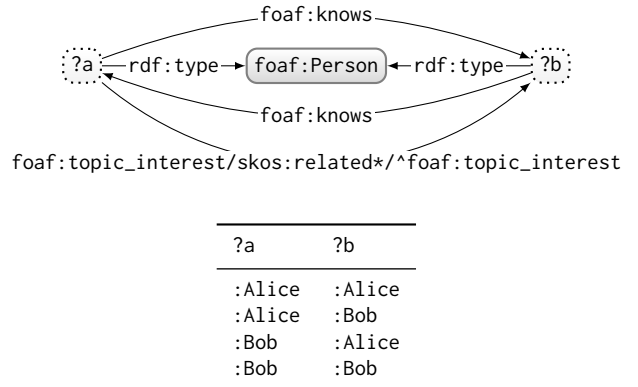| ?a	?b
:Alice	:Alice
:Alice	:Bob
:Bob	:Alice
:Bob	:Bob

Fig. 3: Example NGP (above) and its evaluation over the graph of Figure 1 (below)

where variables are attributes (i.e., column names) and tuples (i.e., rows) contain the RDF terms bound by each solution mapping (see Figures 2–4). CGPs support combining and transforming the results of BGPs/NGPs with language features that include FILTER (selection: σ), SELECT (projection: π), UNION (union: $\cup$), EXISTS (semi-join: $\ltimes$), MINUS (anti-join: $\rhd^4$) and OPTIONAL (left-join: $⟗$). These language features correspond to the relational algebra defined in Table 3. The default operator is a natural inner join ($\bowtie$). Figure 4 provides an example of a CGP combining two BGPs and an NGP using union, join and projection.

Table 3: Core relational algebra of SPARQL

$$\sigma_R(M) := \{ \mu \in M \mid R(\mu) \}$$
$$\pi_V(M) := \{ \mu' \mid \exists \mu \in M : \mu \sim \mu' \wedge \mathrm{dm}(\mu') = V \cap \mathrm{dm}(\mu) \}$$
$$M_1 \bowtie M_2 := \{ \mu_1 \cup \mu_2 \mid \mu_1 \in M_1 \wedge \mu_2 \in M_2 \wedge \mu_1 \sim \mu_2 \}$$
$$M_1 \cup M_2 := \{ \mu \mid \mu \in M_1 \vee \mu \in M_2 \}$$
$$M_1 \ltimes M_2 := \{ \mu_1 \in M_1 \mid \exists \mu_2 \in M_2 : \mu_1 \sim \mu_2 \}$$
$$M_1 \rhd M_2 := \{ \mu_1 \in M_1 \mid \nexists \mu_2 \in M_2 : \mu_1 \sim \mu_2 \}$$
$$M_1 ⟗ M_2 := (M_1 \bowtie M_2) \cup (M_1 \rhd M_2)$$

Named graphs SPARQL allows for querying multiple RDF graphs through the notion of a *SPARQL dataset*, defined as $D := \{ G, (n_1, G_1), \ldots, (n_k, G_k) \}$ where $G, G_1 \ldots, G_n$ are RDF graphs; $n_1, \ldots, n_k$ are pairwise distinct IRIs; G
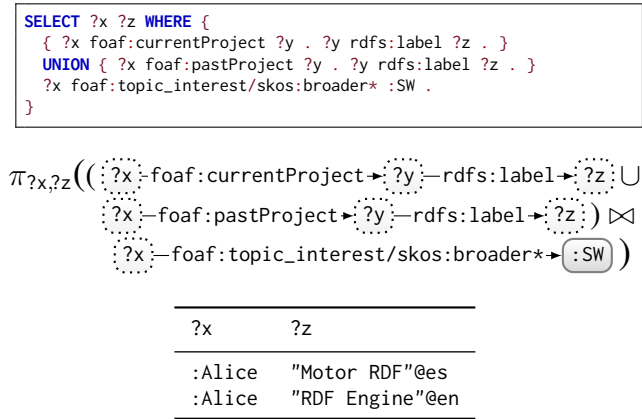
---

[4] The definition of MINUS is slightly different from anti-join in that mappings with no overlapping variables on the right are ignored.

```
SELECT ?x ?z WHERE {
  { ?x foaf:currentProject ?y . ?y rdfs:label ?z . }
  UNION { ?x foaf:pastProject ?y . ?y rdfs:label ?z . }
  ?x foaf:topic_interest/skos:broader* :SW .
}
```

$$\pi_{?x,?z}\Big(\big(\boxed{?x} \!-\! \texttt{foaf:currentProject} \!\rightarrow\! \boxed{?y} \!-\! \texttt{rdfs:label} \!\rightarrow\! \boxed{?z} \; \cup$$
$$\boxed{?x} \!-\! \texttt{foaf:pastProject} \!\rightarrow\! \boxed{?y} \!-\! \texttt{rdfs:label} \!\rightarrow\! \boxed{?z} \big) \bowtie$$
$$\boxed{?x} \!-\! \texttt{foaf:topic\_interest/skos:broader*} \!\rightarrow\! \boxed{:SW} \Big)$$

?x	?z
:Alice	"Motor RDF"@es
:Alice	"RDF Engine"@en

Fig. 4: Example CGP (above) and its evaluation over the graph of Figure 1 (below)

is known as the *default graph*; and each pair (n_1, G_1) (for $1 \le i \le n$) is known as a *named graph*. Letting N', N'' denote sets of IRIs, n', n'' IRIs and v a variable, SPARQL then provides a number of features for querying different graphs:

- FROM N' FROM NAMED N'': activates a dataset with a default graph composed of the merge of all graphs G' such that $(n', G') \in D$ and $n' \in N'$, and the set of all named graphs $(n'', G'') \in D$ such that $n'' \in N''$;
- GRAPH n': evaluates a graph pattern on the graph G' if the named graph (n', G') is active;
- GRAPH v: takes the union of the evaluation of a graph pattern over each G' such that (n', G') is active, binding v to n' for each solution generated from G';

Without FROM or FROM NAMED, the active dataset is the indexed dataset D. Without GRAPH, graph patterns are evaluated on the active default graph. *Quad stores* disallow empty named graphs, such that $D := \{G, (n_1, G_1), \ldots, (n_k, G_k)) \}$ is viewed as $D = G \times \{\star\} \cup (\bigcup_{(n_i, G_i) \in D} G_i \times \{n_i\})$, i.e., a set of quads using $\star \notin \mathbf{IBL}$ as a special symbol for the default graph. In this case, a quad (s, p, o, n) denotes a triple (s, p, o) in the default graph if $n = \star$, or a triple in the named graph G' such that $(n, G') \in D$ if $n \in \mathbf{I}$. We can define CGPs involving quad patterns analogously.

Other SPARQL features SPARQL supports features beyond CGPs, which include aggregation (group-by with count, sum, etc.), solution modifiers (ordering and slicing solutions), bag semantics (preserving result multiplicity), federation (fetching solutions from remote services), entailment and more besides. SPARQL also supports different query types, such as SELECT, which returns a sequence of solution mappings; CONSTRUCT, which returns an RDF graph based on the solution mappings; DESCRIBE, which returns an RDF graph describing indicated RDF terms; and ASK, which returns true if some solution mapping is found, or false otherwise.

4 Storage

Data storage refers to how data are represented in memory. Different storage mechanisms store different elements of data contiguously in memory, offering trade-offs in terms of compression and efficient data access. This section reviews various categories of RDF storage.

4.1 Triple table

A *triple table* stores an RDF graph G as a single ternary relation. Figure 1 shows an RDF graph with its triple table on the right-hand side. One complication when storing triple tables in relational databases is that such systems assume a column to have a single type, which may not be true for RDF objects in particular; a workaround is to store a string encoding of the terms, though this may complicate their ordering.

Rather than storing full RDF terms in the triple table, stores may apply dictionary encoding, where RDF terms are mapped one-to-one with numeric object identifiers (OIDs), with OIDs being stored in the table and decoded using the dictionary as needed. Since OIDs consume less memory and are faster to process than strings, such an approach works better for queries that involve many intermediate results but generate few final results; on the other hand, such an approach suffers when queries are simple and return many results, or when selective filters are specified that require decoding the term before filtering. To find a better trade-off, some RDF engines (e.g., Jena 2 [241]) only use OIDs for strings with lengths above a threshold.

The most obvious physical storage is to store triples contiguously (row-wise). This allows for quickly retrieving the full triples that match (e.g.) a given triple pattern. However, some RDF engines based on relational storage (e.g., Virtuoso [69]) rather use (or provide an option for) column-wise storage, where the values along a column are stored contiguously, often following a particular order. Such column-wise storage allows for better compression, and for quickly reading many values from a single column.

Triple tables can be straightforwardly extended to quad tables in order to support SPARQL datasets [69, 91].

4.2 Vertical partitioning

The *vertical partitioning* approach [1] uses a binary relation for each property $p \in \mathrm{p}(G)$ whose tuples encode subject–object pairs for that property. In Figure 5 we exemplify two such binary relations. Physical storage can again use OIDs, row-based or column-based storage, etc.

When compared with triple tables, vertical partitioning generates relations with fewer rows, and more specific domains for columns (e.g., the object column for foaf:age

rdf:type

Subject	Object
:Alice	foaf:Person
:Bob	foaf:Person
_:p	foaf:Project

foaf:age

Subject	Object
:Alice	26
:Bob	21

Fig. 5: Vertical partitioning for two properties in Figure 1

skos:broader $\ltimes_{\text{S-S}}$ skos:related

Subject	Object
:SW	:Web

skos:broader $\ltimes_{\text{S-O}}$ skos:related

Subject	Object
:DB	:CS

skos:related $\ltimes_{\text{O-S}}$ skos:broader

Subject	Object
:SW	:DB

skos:related $\ltimes_{\text{S-S}}$ skos:broader

Subject	Object
:SW	:DB

Fig. 6: Example semi-join reduction for two properties

foaf:Person

Subject	age	topic	knows	cProj	pProj
:Alice	26	:DB	:Bob	_:p	NULL
:Bob	21	:DB	:Alice	NULL	_:p

Fig. 7: Example property table for people

can be defined as an integer type). However, triple patterns with variable predicates may require applying a union on all relations. Also, RDF graphs may have thousands of properties [233], which may lead to a schema with many relations.

Vertical partitioning can be used to store quads by adding a **Graph** column to each table [69, 91].

4.3 Extended vertical partitioning

S2RDF [204] uses *extended vertical partitioning* based on *semi-join reductions* (we recall from Table 3 that a semi-join $M_1 \ltimes M_2$, aka. FILTER EXISTS, returns the tuples in M_1 that are "joinable" with M_2). Letting $\mathbf{x}, \mathbf{y}, \mathbf{z}$ denote variables and p, q denote RDF terms, then for each property pair $(p, q) \in$ p$(G) \times$ p(G) such that $p \neq q$, extended vertical partitioning stores three semi-join reductions:

1. $(\mathbf{x}, p, \mathbf{y})(G) \ltimes (\mathbf{y}, q, \mathbf{z})(G)$ (O–S),
2. $(\mathbf{x}, p, \mathbf{y})(G) \ltimes (\mathbf{x}, q, \mathbf{z})(G)$ (S–S),
3. $(\mathbf{x}, p, \mathbf{y})(G) \ltimes (\mathbf{z}, q, \mathbf{x})(G)$ (S–O).

The semi-join $(\mathbf{x}, p, \mathbf{y})(G) \ltimes (\mathbf{z}, q, \mathbf{y})(G)$ (O–O) is not stored as most O–O joins have the same predicate, and thus would occur in the same relation. In Figure 6 we give an example of a semi-join reduction for two predicates from the running example; empty semi-joins are omitted.

In comparison with vertical partitioning, observing that $(M_1 \ltimes M_2) \bowtie (M_2 \ltimes M_1) \equiv M_1 \bowtie M_2$, we can apply joins over the corresponding semi-join reductions knowing that each tuple read from each side will contribute to the join, thus reducing I/O. The cost involves storing (and updating) each tuple in up to $3(|$p$(G)| - 1)$ additional relations; omitting empty semi-joins can help to mitigate this issue [204]. Extended vertical partitioning also presents complications for variable predicates, graphs with many properties, etc.

4.4 Property table

Property tables aim to emulate the n-ary relations typical of relational databases. A property table usually contains one subject column, and n further columns to store objects for the corresponding properties of the given subject. The subject column then forms a primary key for the table. The tables to define can be based on classes, clustering [184], coloring [36], etc., to group subjects with common properties.

We provide an example of a property table based on the class foaf:Person in Figure 7 for the RDF graph of Figure 1.

Property tables can store and retrieve multiple triples with a given subject as one tuple (e.g., to find people with **age** < 30 and **interest** $=$:SW) without needing joins. Property tables often store terms of the same type in the same column, enabling better compression. Complications arise for multi-valued (. . .-to-many) or optional (zero-to-. . .) properties. In the example of Figure 1, Alice is also interested in SW, which does not fit in the cell. Furthermore, Alice has no past project, and Bob has no current project, leading to nulls. Changes to the graph may also require re-normalization; for example, even though each person currently has only one value for **knows**, adding that Alice knows another person would require renormalizing the tables. Complications also arise when considering variable predicates, RDF graphs with many properties or classes, quads, etc.

4.5 Graph-based storage

While the previous three storage mechanisms rely on relational storage, *graph-based storage* is adapted specifically for the graph-based model of RDF. Key characteristics of such models that can be exploited for storage include the *adjacency* of nodes, the fixed *arity* of graphs, etc.

Graphs have bounded arity (3 for triples, 4 for quads), which can be exploited for specialized storage. Engines like 4store [91] and YARS2 [94] build *native triple/quad tables*, which differ from relational triple/quad tables in that they have fixed arity, fixed attributes (S,P,O(,G)), and more general domains (e.g., the O column can contain any RDF term).

Graphs often feature local repetitions that are compressible with *adjacency lists* (e.g., Hexastore [238], gStore [263],
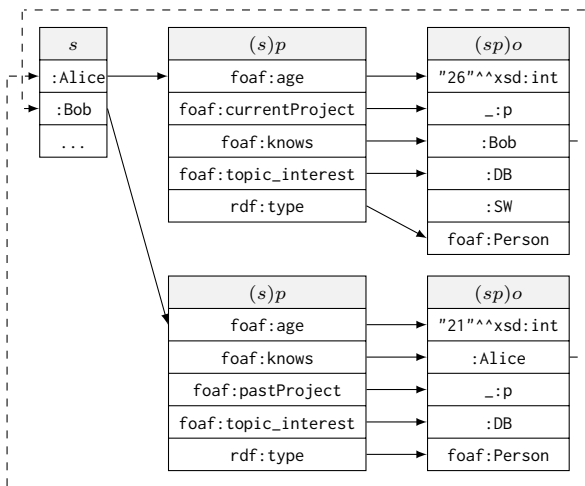
Fig. 8: Example adjacency list for two subjects with dashed links indicating index-free adjacency pointers

Fig. 9: Example of the multi-graph representation

Fig. 10: Example bit matrix for skos:broader

4.6 Tensor-based storage

Another type of native graph storage uses tensors, viewing a dictionary-encoded RDF graph G with $m = |\text{so}(G)|$ nodes and $n = |\text{p}(G)|$ predicates as an $m \times n \times m$ 3-order tensor $\mathfrak{T}$ of bits such that $\mathfrak{T}_{i,j,k} = 1$ if the ith node links to the kth node with the jth property, or $\mathfrak{T}_{i,j,k} = 0$ otherwise. A popular variant uses an adjacency matrix per property (e.g., BitMat [21], BMatrix [38], QDags [163]), akin to vertical partitioning, as seen in Figure 10. A third option (considered, e.g., by MAGiQ [109]) is to encode the full graph as an adjacency matrix where each cell indicates the property id connecting the two nodes; this matrix cannot directly represent pairs of nodes connected by more than one property.

While abstract tensor-based representations may lead to highly-sparse matrices or tensors, compact data structures offer compressed representations that support efficient operations [21, 109, 38, 163]. Often such matrices/tensors are stored in memory, or loaded into memory when needed. Such representations may also enable query processing techniques that leverage hardware acceleration, e.g., for processing joins on GPUs (as we will discuss in Section 6.4).

4.7 Miscellaneous storage

Aside from relational-based and graph-based storage, other engines have proposed to leverage other forms of storage as implemented by existing systems. A common example is the use of NoSQL key-value, tabular or document stores for distributed storage (see [111, 257, 249] for more details).

4.8 Discussion

Early works on storing RDF tended to rely on relational storage, which had been subject to decades of developments and optimizations before the advent of RDF (e.g., [241, 1, 69]). Though such an approach still has broad adoption [69], more recent storage techniques aim to exploit the graph-based characteristics of RDF – and SPARQL – in order to develop dedicated storage techniques (e.g., [21, 238, 263]), including those based on tensors/matrices [21, 109, 38, 163]. A recent trend is to leverage NoSQL storage (e.g., [131, 177, 25]) in order to distribute the management of RDF data.

SpiderStore [32], Trinity.RDF [258], GRaSS [142]). These lists are akin to tries, where subject or subject–predicate prefixes are followed by the rest of the triple. Such tries can be stored row-wise in blocks of triples; or column-wise, where blocks elements from one column point to blocks of elements from the next column. *Index-free adjacency* can enable efficient navigation, where terms in the suffix directly point to the location on disk of their associated prefix. We refer to Figure 8 for an example. Such structures can also include inverse edges (e.g., Trinity.RDF [258], GRaSS [142]).

An alternative is to decompose an RDF graph into its constituent components for storage. AMBER [105] uses a *multigraph representation* where an RDF graph G is decomposed into a set of (non-literal) nodes $V := \text{so}(G) \cap \mathbf{IB}$, a set of edges $E := \{(s, o) \in V \times V \mid \exists p : (s, p, o) \in G\}$, an edge-labeling function of the form $L_E : V \to 2^{\mathbf{I}}$ such that $L_E(s, o) := \{p \mid (s, p, o) \in G\}$, and an attribute-labeling function of the form $L_V : \mathbf{IB} \to 2^{\mathbf{I} \times \mathbf{L}}$ such that $L_V(s) := \{(p, o) \mid (s, p, o) \in G \wedge o \in \mathbf{L}\}$, as seen in Figure 9 (in practice, AMBER uses dictionary-encoding).

5 Indexing

Indexing enables efficient lookup operations on RDF graphs (i.e., $O(1)$ or $O(\log |G|)$ time to return the first result or an empty result). The most common such operation is to find triples that match a given triple pattern. However, indexes can also be used to match non-singleton BGPs (with more than one triple pattern), to match path expressions, etc. We now discuss indexing techniques proposed for RDF graphs.

5.1 Triple indexes

The goal of *triple indexes* is to efficiently find triples matching a triple pattern. Letting s, p, o denote RDF terms and $\mathbf{s}, \mathbf{p}, \mathbf{o}$ variables, there are $2^3 = 8$ abstract patterns: $(\mathbf{s}, \mathbf{p}, \mathbf{o})$, $(\mathbf{s}, \mathbf{p}, o)$, $(\mathbf{s}, p, \mathbf{o})$, $(s, \mathbf{p}, \mathbf{o})$, $(\mathbf{s}, p, o)$, $(s, \mathbf{p}, o)$, $(s, p, \mathbf{o})$ and (s, p, o). Unlike relational databases, where often only the primary key of a relation will be indexed by default and further indexes must be manually specified, most RDF stores aim to have a complete index by default, covering all eight possible triple patterns. However, depending on the type of storage chosen, this might not always be feasible.

When a storage scheme such as vertical partitioning is used [1], only the five patterns where the predicate is constant can be efficiently supported (by indexing the subject and object columns). If the RDF graph is stored as a (binary) adjacency matrix for each property [21,163], again only constant-predicate patterns can be efficiently supported. Specialized indexes can be used to quickly evaluate such patterns, where QDags [163] uses *quadtrees*: a hierarchical index structure that recursively divides the matrix into four sub-matrices; we provide an example quadtree in Figure 11. A similar structure, namely a k^2-*tree*, is used by BMatrix [38].

Otherwise, in triple tables, or similar forms of graph-based storage, all triple patterns can be efficiently supported with *triple permutations*. Figure 8 illustrates a single SPO permutation. A total of $3! = 6$ permutations are possible and suffice to cover all eight abstract triple patterns if the index structure permits prefix lookups; for example, in an SPO permutation we can efficiently support four abstract triple patterns $(\mathbf{s}, \mathbf{p}, \mathbf{o})$, $(s, \mathbf{p}, \mathbf{o})$, $(s, p, \mathbf{o})$ and (s, p, o) as we require the leftmost terms of the permutation to be filled. In fact, with only $\binom{3}{\lfloor 3/2 \rfloor} = 3$ permutations – e.g., SPO, POS and OSP – we can cover all eight abstract triple patterns. Such index permutations can be implemented using standard data structures such as ISAM files [94], B(+)Trees [168], AVL trees [243], as well as compact data structures, such as adjacency lists [238] (see Figure 8) and tries [185], etc.

Recent works use compact data structures to reduce redundancy for index permutations, and thus the space required for triple indexing. Perego et al. [185] use tries to
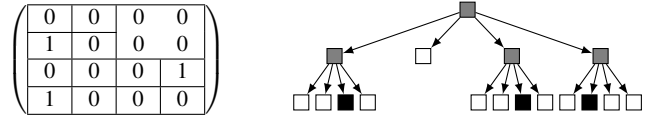

Fig. 11: A quadtree index based on the bit matrix of Figure 10; the root represents the full matrix, while children denote four sub-matrices of the parent; a node is colored black if it contains only 1's, white if it contains only 0's, and gray if it contains both; only gray nodes require children

index multiple permutations, over which they apply *cross-compression*, whereby the order of the triples given by one permutation is used to compress another permutation. Other approaches remove the need for multiple permutations. RDFCSA [40] and Ring [19] use a *compact suffix-array* (*CSA*) such that one permutation suffices to efficiently support all triple patterns. Intuitively speaking, triples can be indexed cyclically in a CSA, such that in an SPO permutation, one can continue from O back to S, thus covering SPO, POS and OSP permutations in one CSA index [40]. The Ring indexing scheme is also bidirectional, where in an SPO permutation, one can move from O forwards to S or backwards to P.

5.2 Entity-based indexes

Entity-based indexes optimize graph patterns that "center on" a particular entity. BGPs can be reduced to joins over their triple patterns; for example, $\{(\mathbf{x}, p, \mathbf{y}), (\mathbf{y}, q, \mathbf{z})\}(G) = \{(\mathbf{x}, p, \mathbf{y})\}(G) \bowtie \{(\mathbf{y}, p, \mathbf{z})\}(G)$. *Star joins* are frequently found in BGPs, defined to be a join on a common subject, e.g., $\{(\mathbf{w}, p, \mathbf{x}), (\mathbf{w}, q, \mathbf{y}), (\mathbf{w}, r, z)\}$. Star joins may sometimes also include S–O joins on the common variable, e.g., $\{(\mathbf{w}, p, \mathbf{x}), (\mathbf{w}, q, \mathbf{y}), (z, r, \mathbf{w})\}$ [142]. Star joins retrieve data surrounding a particular entity (in this case $\mathbf{w}$). Entity-based indexes permit efficient evaluation of such joins.

Property tables can enable efficient star joins so long as the relevant tables can be found efficiently and there are indexes on the relevant columns (e.g., for p, q and/or r).

The EAGRE system [261] uses an index for property tables where entities with n properties are encoded in n-dimensional space. A space-filling curve (e.g., a Z-order or Hilbert curve) is then used for indexing. Figure 12 illustrates the idea, where four entities are indexed (abbreviating :Alice, :Bob, :Carol, :Dave) with respect to two dimensions (say foaf:age for x and integer-encoded values of foaf:knows for y). We show the first-, second- and third-order Hilbert curves from left to right. Letting d denote the number of dimensions, the nth-order Hilbert curve assigns an ordinal to 2^{dn} regions of the space based on the order in which it visits the region; e.g., starting with region 1 on the bottom left and following the curve, :A is in the region of ordinal 2, 7 and 26, respectively. The space-filling curve thus
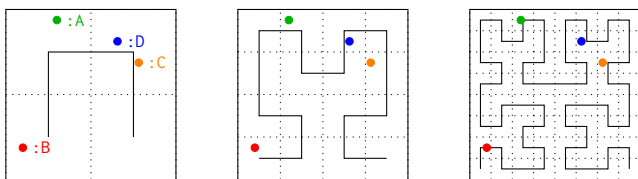
Fig. 12: Space-filling indexing with a Hilbert curve

Fig. 13: Lattice of node signatures with abbreviated terms (e.g., `s:b` denotes `skos:broader`)

"flattens" multidimensional data into one dimension (the ordinal), which can be indexed sequentially.

Property tables are complicated by multi-valued properties, missing values, etc. A more flexible approach is to index *signatures* of entities, which are bit-vectors encoding the property–value pairs of the entity. One such example is the *vertex signature tree* of gStore [263], which encodes all outgoing (p, o) pairs for a given entity s into a bit vector akin to a Bloom filter, and indexes these bit vectors hierarchically allowing for fast, approximate containment checks that quickly find candidate entities for a subset of such pairs. GRaSS [142] further optimizes for star subgraphs that include both outcoming and incoming edges on entities, where a custom *FDD-index* allows for efficient retrieval of the subgraphs containing a triple that matches a triple pattern.

5.3 Property-based indexes

Returning to the star join $\{(\mathbf{w}, p, \mathbf{x}), (\mathbf{w}, q, \mathbf{y}), (\mathbf{w}, r, z)\}$, another way to quickly return candidate bindings for the variable $\mathbf{w}$ is to index nodes according to their adjacent properties; then we can find nodes that have at least the adjacent properties p, q, r. Such an approach is used by RDFBroker [212], which defines the signature of a node s as $\Sigma(s) = \{p \mid \exists o : (s, p, o) \in G\}$; for example, the signature of `:SW` in Figure 1 is $\Sigma(\texttt{:SW}) = \{\texttt{skos:broader}, \texttt{skos:related}\}$ (analogous to *characteristic sets* proposed later [165]). A property table is then created for each signature. At query time, property tables whose signatures subsume $\{p, q, r\}$ are found using a lattice of signatures. We provide an example in Figure 13 with respect to the RDF graph of Figure 1, where children subsume the signatures of their parent.

AxonDB [155] uses *extended characteristic sets* where each triple (s, p, o) in the RDF graph is indexed with the signatures (i.e., characteristic sets) of its subject and object; i.e., $(\Sigma(s), \Sigma(o))$. Thus the triple (`:SW, skos:related, :DB`) of Figure 1 would be indexed with the extended characteristic set ($\{\texttt{skos:broader}, \texttt{skos:related}\}$, $\{\texttt{skos:broader}\}$). The index then allows for efficiently identifying two star joins that are connected by a given property p.
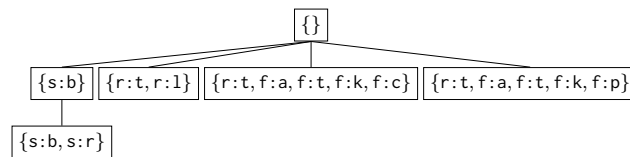
5.4 Path indexes

A path join involves successive s–o joins between triple patterns; e.g., $\{(\mathbf{w}, p, \mathbf{x}), (\mathbf{x}, q, \mathbf{y}), (\mathbf{y}, r, z)\}$, where the start and end nodes ($\mathbf{w}, z$) may be variables or constants. While path joins have fixed length, navigational graph patterns may further match arbitrary length paths. A number of indexing approaches have been proposed to speed up querying paths.

A path can be seen as a string of arbitrary length; e.g., a path $\{(w, p, x), (x, q, y), (y, r, z)\}$ can be seen as a string $wpxqyrz\$$, where $\$$ indicates the end of the string; alternatively, if intermediate nodes are not of importance, the path could be represented as the string $wpqrz\$$. The Yaanii system [47] builds an index of paths of the form $wpxqyrz\$$ that are clustered according to their template of the form $wpqrz\$$. Paths are then indexed in B+trees, which are partitioned by template. Fletcher et al. [72] also index paths in B+trees, but rather than partition paths, they apply a maximum length of at most k for the paths included. Text indexing techniques can also be applied for paths (viewed as strings). Maharjan et al. [147] and the HPRD system [139] both leverage suffix arrays – a common indexing technique for text – to index paths. The downside of path indexing approaches is that they may index an exponential number of paths; in the case of HPRD, for example, users are thus expected to specify which paths to index [139].

Other path indexes are inspired by prior works for path queries over trees (e.g., for XPath). Bartoň [26] proposes a tree-based index based on *preorder* and *postorder traversal*. A preorder traversal starts at the root and traverses children in a depth-first manner from left to right. A postorder traversal starts at the leftmost leaf and traverses all children, from left to right, before moving to the parent. We provide an example preorder and postorder traversal in Figure 14. Given two nodes m and n in the tree, a key property is that m is a descendant of n if and only if m is greater than n for preorder and less than n for postorder. Bartoň [26] uses this property to generate an index on ascending preorder so as to linearize the tree and quickly find descendants based on postorder. To support graphs, Bartoň uses a decomposition of the graph into a forest of trees that are then indexed [26].

Another type of path index, called *PLSD*, is used in System Π [245] for indexing the transitivity of a single property, optimizing for path queries of the form $(\mathbf{s}, p*, \mathbf{o})$, or
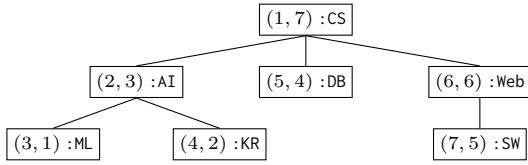
Fig. 14: Preorder and postorder on a `skos:narrower` tree; e.g., `:CS` has preorder 1 and postorder 7

Fig. 15: PLSD index on an example `skos:narrower` hierarchy; terms (e.g., `:CS`) are indexed externally

Fig. 16: Distance-based indexing (GRIN)

$(s, p*, \mathbf{o})$, etc. For a given property p, each incident (subject or object) node x is assigned a triple of numbers $(i, j, k) \in \mathbb{N}^3$, where i is a unique prime number that identifies the node x, j is the least common multiple of the i-values of x's parents (i.e., nodes y such that $(y, p, x) \in G$), and k is the least common multiple of the k-values of x's parents and the i-value of x. We provide an example in Figure 15. PLSD can further handle cycles by multiplying the k-value of all nodes by the i value of all nodes in its strongly-connected component. Given the i-value of a node, the i-values of its parents and ancestors can be retrieved by factorizing j and k/i respectively. However, multiplication may give rise to large numbers, where no polynomial time algorithm is known for the factorization of binary numbers.

Gubichev et al. [80] use a path index of directed graphs, called FERRARI [210], for each property in an RDF graph. First, a *condensed graph* is computed by merging nodes of strongly connected components into one "supernode"; adding an artificial root node (if one does not exist), the result is a directed acyclic graph (DAG) that preserves reachability. A *spanning tree* – a subgraph that includes all nodes and is a tree – of the DAG is computed and labeled with its postorder. All subtrees thus have contiguous identifiers, where the maximum identifies the root; e.g., in Figure 14, the subtree at `:AI` has the interval $[1, 3]$, where 3 identifies the root. Then there exists a (directed) path from x to y if and only if y is in the subtree interval for x. Nodes in a DAG may, however, be reachable through paths not in the spanning tree. Hence each node is assigned a *set* of intervals for nodes that can be reached from it, where overlapping and adjacent intervals are merged; we must now check that y is in one of the intervals of x. To improve time and space at the cost of precision, approximate intervals are proposed that merge non-overlapping intervals; e.g., $[4, 6]$, $[8, 9]$ is merged to $[4, 9]$, which can reject reachability for nodes with id less than 2 or greater than 9, but has a $\frac{1}{6}$ chance of a false positive for nodes in $[4, 9]$, which must be verified separately.

5.5 Join indexes

The results of joins can also be indexed. Groppe et al. [78] proposed to construct $6 \times 2^4 = 96$ indexes for 6 types of non-symmetric joins between two triple patterns (S–S, S–
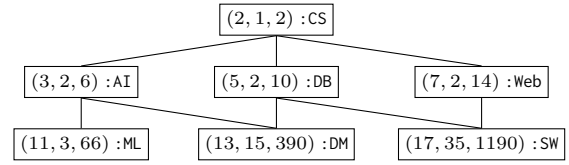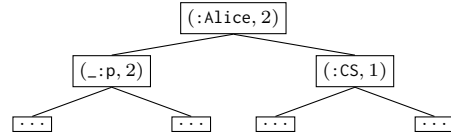
P, S–O, P–P, P–O, O–O). Hash maps are used to cover the 2^4 permutations of the remaining elements (not considering the join variable). Given the high space cost, only frequently encountered joins are sometimes indexed [55, 152].

5.6 Structural indexes

Another family of indexes – known as *structural indexes* [141] – rely on a high-level summary of the RDF graph.

Some structural indexes are based on distance measures. GRIN [228] divides the graph hierarchically into regions based on the distance of its nodes to selected centroids. These regions form a tree, where the non-leaf elements indicate a node x and a distance d referring to all nodes at most d steps from x. The root element chooses a node and distance such that all nodes of the graph are covered. Each non-leaf element has two children that capture all nodes of their parent. Each leaf node contains a set of nodes N, which induces a subgraph of triples between the nodes of N; the leaves can then be seen as partitioning the RDF graph. We provide an example in Figure 16 for the RDF graph of Figure 1, where all nodes are within distance two of `:Alice`, which are then divided into two regions: one of distance at most two from `_:p`, and another of distance at most one from `:CS`. The index can continue dividing the graph into regions, and can then be used to find subgraphs within a particular distance from a given node (e.g., a node given in a BGP).

Another type of structural index relies on some notion of a *quotient graph* [48], where the nodes of a graph $\mathrm{so}(G)$ are partitioned into $\{X_1, \dots, X_n\}$ pairwise-disjoint sets such that $\bigcup_{i=1}^{n} X_i = \mathrm{so}(G)$. Then edges of the form (X_i, p, X_j) are added if and only if there exists $(x_i, p, x_j) \in G$ such that $x_i \in X_i$ and $x_j \in X_j$. Intuitively, a quotient graph merges nodes from the input graph into "supernodes" while maintaining the input (labeled) edges between the supernodes. We provide an example of a quotient graph in Figure 17 featuring six supernodes. Any partitioning of nodes can form
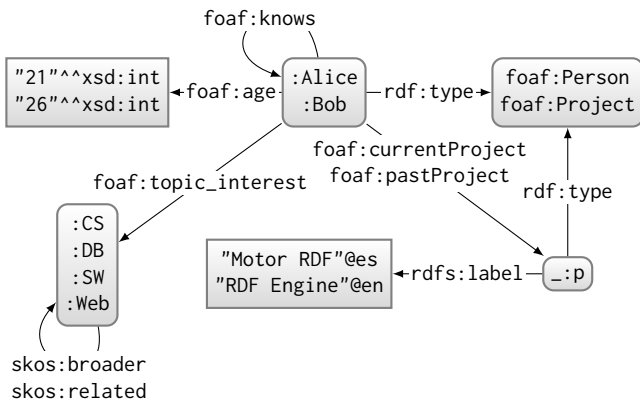
Fig. 17: Quotient graph with six supernodes

a quotient graph, ranging from a single supernode with all nodes $\mathrm{so}(G)$ and loops for all properties in $\mathrm{p}(G)$, to the graph itself replacing each node $x \in \mathrm{so}(G)$ with the singleton $\{x\}$. If the input graph yields solutions for a BGP, then the quotient graph will also yield solutions (with variables now matching supernodes). For example, taking the BGP of Figure 2, matching `foaf:Person` to the supernode containing `foaf:Person` in Figure 17, then the variables ?a and ?b will match the supernode containing `:Alice` and `:Bob`, while ?ia and ?ib will match to the supernode containing `:CS`, `:DB`, `:SW` and `:Web`; while we do not know the exact solutions for the input graph, we know they must correspond to elements of the supernodes matched in the quotient graph.

DOGMA [41] partitions an RDF graph into subgraphs, from which a balanced binary tree is computed, where each parent node contains a quotient-like graph of both its children. The (O)SQP approach [225] creates an in-memory index graph, which is a quotient graph whose partition is defined according to various notions of bisimulation.

SAINT-DB [186] adopts a similar approach, where supernodes are defined directly as a partition of the triples of the RDF graph, and edges between supernodes are labeled with the type of join (S–S, P–O, etc.) between them.

5.7 Quad indexes

Most quad indexes follow the triple index scheme [243, 94, 91, 69], extending it to add another element. The number of permutations then grows to $2^4 = 16$ abstract index patterns, $4! = 24$ potential permutations, and $\binom{4}{\lfloor 4/2 \rfloor} = 6$ flat (ISAM/B+Tree/AVL tree/trie) permutations or 2 circular (CSA) permutations to efficiently support all abstract quad patterns. A practical compromise is to maintain a selection of permutations that cover the most common patterns [69]; for example, a pattern $(s, \mathbf{p}, o, g)$ may be uncommon in prac-

tice, and could be supported reasonably well by evaluating (e.g.) $(s, \mathbf{p}, o, \mathbf{g})$ and filtering on $\mathbf{g} = g$.

The RIQ system [120] proposes a custom index for quads called a PV-index for finding (named) graphs that match a BGP. Each graph is indexed by hashing all seven abstract patterns on triples with some constant, generating seven *pattern vectors* for each graph. For example, a triple (s, p, o) in a graph named g will be hashed as (s, p, o), $(s, p, ?)$, $(s, ?, o)$, $(?, p, o)$, $(s, ?, ?)$, $(?, p, ?)$, $(?, ?, o)$, where $?$ is an arbitrary fixed token, and each result will be added to one of seven pattern vectors for g for that abstract pattern. Basic graph patterns can be encoded likewise, where locality sensitive hashing is then used to group and retrieve similar pattern vectors for a given basic graph pattern.

5.8 Miscellaneous Indexing

RDF stores may use legacy systems, such as NoSQL stores, for indexing. Since such approaches are not tailored to RDF, and often correspond conceptually to one of the indexing schemes already discussed, we refer to more dedicated surveys of such topics for further details [111, 257, 249]. Other stores provide specialized indexes for particular types of values such as spatial or temporal data [232, 130]; we do not discuss such specialized indexes in detail.

5.9 Discussion

While indexing triples or quads is conceptually the most straightforward approach, a number of systems have shown positive results with entity- and property-based indexes that optimize the evaluation of star joins, path indexes that optimize the evaluation of path joins, or structural indexes that allow for identifying query-relevant regions of the graph. Different indexing schemes often have different time–space trade-offs: more comprehensive indexes enable faster queries at the cost of space and more costly updates.

6 Join Processing

RDF stores employ diverse query processing strategies, but all require translating logical operators that represent the query, into "physical operators" that implement algorithms for efficient evaluation of the operation. The most important such operators – as we now discuss – are natural joins.

6.1 Pairwise join algorithms

We recall that the evaluation of a BGP $\{t_1, \ldots t_n\}(G)$ can be rewritten as $t_1(G) \bowtie \ldots \bowtie t_n(G)$, where the evaluation

of each triple pattern t_i ($1 \leq i \leq n$) produces a relation of arity $|\text{vars}(t_i)|$. Thus the evaluation of a BGP B produces a relation of arity $|\text{vars}(B)|$. The relational algebra – including joins – can then be used to combine or transform the results of one or more BGPs, giving rise to CGPs. The core of evaluating graph patterns is thus analogous to processing relational joins. The simplest and most well-known such algorithms perform *pairwise joins*; for example, a pairwise strategy for computing $\{t_1, \ldots t_n\}(G)$ may evaluate $((t_1(G) \bowtie t_2(G)) \bowtie \ldots) \bowtie t_n(G)$.

Without loss of generality, we assume a join of two graph patterns $P_1(G) \bowtie P_2(G)$, where the join variables are denoted by $V = \{v_1, \ldots, v_n\} = \text{vars}(P_1) \cap \text{vars}(P_2)$. Well-known algorithms for performing pairwise joins include *(index) nested-loop joins*, where $P_1(G) \bowtie P_2(G)$ is reduced to evaluating $\bigcup_{\mu \in P_1(G)} \{\mu\} \bowtie \mu(P_2)(G)$; *hash joins*, where each solution $\mu \in P_1(G)$ is indexed by hashing on the key $(\mu(v_1), \ldots, \mu(v_n))$ and thereafter a key is computed likewise for each solution in $P_2(G)$ to probe the index with; and *(sort-)merge joins*, where $P_1(G)$ and $P_2(G)$ are (sorted if necessary and) read in the same order with respect to V, allowing the join to be reduced to a merge sort. Index nested-loop joins tend to perform well when $|P_1(G)| \ll |P_2(G)|$ (assuming that $\mu(P_2)(G)$ can use indexes) since it does not require reading all of $P_2(G)$. Otherwise hash or merge joins can perform well [168]. Pairwise join algorithms are then used in many RDF stores (e.g., [93,69,168]).

Techniques to optimize pairwise join algorithms include *sideways information passing* [29], which passes data across different parts of the query, often to filter intermediate results. Neumann and Weikum [167] propose *ubiquitous sideways information passing* (*U-SIP*) for computing joins over RDF, which shares global ranges of values for a given query variable. U-SIP is implemented differently for different join types. For merge joins, where data are read in order, a maximum value for a variable can be shared across pairwise joins, allowing individual operators to skip ahead to the current maximum. For hash joins, a global *domain filter* is employed – consisting of a maximum value, a minimum value, and Bloom filters – for filtering the results of each variable.

6.2 Multiway joins

Multiway join algorithms exploit the commutativity and associativity of joins to evaluate two *or more* operands at once. For example, in order to compute $\{t_1, \ldots t_n\}(G)$, a multiway join algorithm may evaluate $(t_1(G) \bowtie \ldots \bowtie t_k(G)) \bowtie (t_{k+1}(G) \bowtie \ldots \bowtie t_n(G))$ where $k \geq 2$, or it may even simply evaluate everything at once as $(t_1(G) \bowtie \ldots \bowtie t_n(G))$.

Some of the previous storage and indexing schemes we have seen lend themselves naturally to processing certain types of multiway joins in an efficient manner. Entity-based indexes allow for processing star joins efficiently, while path

indexes allow for processing path joins efficiently (see Section 5). A BGP can be decomposed into sub-BGPs that can be evaluated per the corresponding multiway join, with pairwise joins being applied across the sub-BGPs; for example: $\{(\mathbf{w}, p, \mathbf{x}), (\mathbf{w}, q, \mathbf{y}), (\mathbf{w}, r, z), (\mathbf{x}, q, \mathbf{y}), (\mathbf{x}, r, z)\}$ may be divided into the sub-BGPs $\{(\mathbf{w}, p, \mathbf{x}), (\mathbf{w}, q, \mathbf{y}), (\mathbf{w}, r, z)\}$ and $\{(\mathbf{x}, q, \mathbf{y}), (\mathbf{x}, r, z)\}$, which are evaluated separately as multiway joins before being themselves joined. Even in the case of (sorted) triple/quad tables, multiway joins can be applied taking advantage of the locality of processing, where, for example, in an SPO index permutation, triples with the same subject will be grouped together. Similar locality can be exploited in distributed settings (see, e.g., SMJoin [74]).

6.3 Worst case optimal joins

A new family of join algorithms have arisen due to the *AGM bound* [22], which puts an upper bound on the number of solutions that can be returned from relational join queries. The result can be adapted straightforwardly to the case of BGPs. Let $B = \{t_1, \ldots, t_n\}$ denote a BGP with $\text{vars}(B) = V$. Now define a *fractional edge cover* as a mapping $\lambda : B \rightarrow \mathbb{R}_{[0,1]}$ that assigns a real value in the interval $[0, 1]$ to each triple pattern of B such that for all $v \in V$, it holds that $\sum_{t \in B_v} \lambda(t) \geq 1$, where B_v denotes the set of triple patterns in B that mention v. The AGM bound tells us that if B has the fractional edge cover λ, then for any RDF graph it holds that $|B(G)| \leq \prod_{i=1}^{n} |t_i(G)|^{\lambda(t_i)}$; this bound is "tight".

To illustrate the AGM bound, consider the BGP $B = \{t_1, t_2, t_3\}$ from Figure 18. There exists a fractional edge cover λ of B such that $\lambda(t_1) = \lambda(t_2) = \lambda(t_3) = \frac{1}{2}$; taking ?a, we have that $B_{?a} = \{t_1, t_3\}$, $\lambda(t_1) + \lambda(t_3) = 1$, and thus ?a is "covered", and we can verify the same for ?b and ?c. Then the AGM bound is given as the inequality $|B(G)| \leq \prod_{i=1}^{n} |t_i(G)|^{\lambda(t_i)}$. For G the graph in Figure 18, $|t_1(G)| = |t_2(G)| = |t_3(G)| = 5$, and hence $|B(G)| \leq 5^{\frac{3}{2}}$. In reality, for this graph, $|B(G)| = 5$, thus satisfying the inequality, but there exists a graph where $B = \prod_{i=1}^{n} |t_i(G)|^{\lambda(t_i)}$.

Recently, join algorithms have been proposed that can enumerate the results for a BGP B over a graph G in time $O(agm(B, G))$, where $agm(B, G)$ denotes the AGM bound of B over G. Since such an algorithm must at least spend $O(agm(B, G))$ time writing the results in the worst case, such algorithms are deemed *worst-case optimal* (*wco*) [169]. Though such algorithms were initially proposed in a relational setting [169, 230], they have recently been adapted for processing joins over RDF graphs [115, 99, 163, 19]. Note that traditional pairwise join algorithms are not wco. If we try to evaluate $\{t_1, t_2\}(G)$ by pairwise join, for example, in order to later join it with $t_3(G)$, the AGM bound becomes quadratic as $\lambda(t_1) = \lambda(t_2) = 1$, and thus we have the bound $|t_1(G)| \cdot |t_2(G)|$, which exceeds the AGM bound for B. This
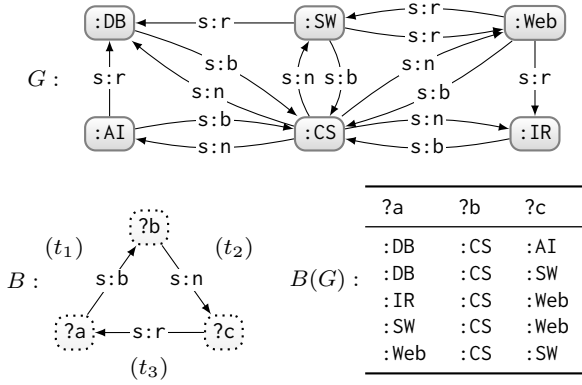
Fig. 18: Example RDF graph G, BGP B and its evaluation $B(G)$; the IRIs s:b, s:n and s:r abbreviate skos:broader, skos:narrower and skos:related, resp.

holds for any pairwise join in B. Note that $\{t_1, t_2\}(G)$ will indeed produce (25) quadratic results, mapping ?a to :CS and ?b and ?c to $\{$:AI, :DB, :IR, :SW, :Web$\}^2$.

Wco join algorithms – including *Leapfrog Triejoin* (LTJ) [230] – perform a multiway join that resolves a BGP B variable-by-variable rather than pattern-by-pattern. First an ordered sequence of variables is selected; say (?a, ?b, ?c). Then the set of partial solutions $M_{\{?a\}} = \{\mu \mid \mathrm{dm}(\mu) = \{?a\}$ and $\mu(B_{?a})(G) \neq \emptyset\}$ are computed for the first variable ?a such that each image of $B_{?a}$ under $\mu \in M_{\{?a\}}$ has some solutions for G; e.g., $M_{\{?a\}} = \{\{?a/$:DB$\}, \{?a/$:IR$\},$ $\{?a/$:SW$\}, \{?a/$:Web$\}\}$ in Figure 18, since replacing ?a in $B_{?a}$ with :DB, :IR, :SW or :Web yields a BGP with solutions over G. Next we compute $M_{\{?a,?b\}} = \{\mu \cup \mu' \mid \mu \in M_{\{?a\}}, \mathrm{dm}(\mu') = \{?b\}$ and $\mu'(\mu(B_{?b}))(G) \neq \emptyset\}$, "eliminating" the next variable ?b. In the example of Figure 18, $M_{\{?a,?b\}} = \{\{?a/$:DB, ?b/:CS$\}, \ldots, \{?a/$:Web, ?b/:CS$\}\}$, where each solution $\mu \in M_{\{?a\}}$ is extended with $\{?b/$:CS$\}$. Finally, $M_{\{?a,?b,?c\}}$ is computed analogously, eliminating the last variable, and yielding the five results seen in Figure 18.

To be wco-compliant, the algorithm must always be able to efficiently compute $M_{\{v\}}$, i.e., solutions μ with $\mathrm{dm}(\mu) = \{v\}$, such that $\mu(B_v)(G) \neq \emptyset$. To compute $M_{\{?a\}}$ in the running example, we need to efficiently intersect all nodes with an outgoing s:b edge and an incoming s:r edge. This is typically addressed by being able to read the results of a triple pattern, in sorted order, for any variable, which enables efficient intersection by allowing to seek ahead to the maximum current value of all triple patterns involving a given variable. Jena-LTJ [99], which implements an LTJ-style join algorithm for SPARQL, enables this by maintaining all six index permutations over triples, while Ring [19] requires only one permutation. Wco algorithms often outperform traditional join algorithms for complex BGPs [115,99].

6.4 Translations to linear algebra

Per Section 4.6, dictionary-encoded RDF graphs are sometimes represented as a bit tensor, or as a bit matrix for each property (see Figure 10), etc. Viewed in this light, some query algebra can then be reduced to linear algebra [156]; for example, joins become matrix/tensor multiplication. To illustrate, we can multiply the bit (adjacency) matrix from Figure 10 for skos:broader by itself:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The result indicates the analogous bit matrix for an O–S join on skos:broader, with :SW (on row 3) connected to :CS (on column 1), which we would expect per Figure 1.

Translating joins into linear algebra enables hardware acceleration, particularly involving GPUs and HPC architectures, which can process tensors with high levels of parallelism. Such an approach is followed by MAGiQ [109], which represents an RDF graph as a single $n \times n$ matrix $\mathfrak{M}$, where n is the number of nodes ($n = |\mathrm{so}(G)|$) and $\mathfrak{M}_{i,j}$ encodes the id of the property connecting the ith node to the jth node (or 0 if no such property exists). One issue with this representation is that it does not support two nodes being connected by multiple edges with different labels, and thus a coordinate list representation can rather be used. Basic graph patterns with projection are translated into matrix multiplication, scalar multiplication, transposition, etc., which can be executed on a variety of hardware, including GPUs.

Other engines that translate SPARQL query features into linear algebra (or other operations within GPUs) include Wukong(+G) [211,235], TripleID-Q [49], and gSmart [52]. Wukong+G [235] proposes a number of caching, pipelining, swapping and prefetching techniques in order to reduce the GPU memory required when processing large graphs while maintaining efficiency, and also proposes a partitioning technique to distribute computation over multiple CPUs and GPUs. TripleID-Q [49] represents an RDF graph as a dictionary-encoded triple table that can be loaded into the GPU in order to search for solutions to individual triple patterns without indexing, but with high degrees of parallelism. On top of this GPU-based search, join and union operators are implemented using GPU libraries. gSmart [52] proposes a variety of optimizations for evaluating basic graph patterns in such settings, including a multi-way join optimization for computing star-like joins more efficiently on GPUs, compact representations for sparse matrices, data partitioning to enable higher degrees of parallelism, and more besides.

6.5 Join reordering

The order of join processing can have a dramatic effect on computational costs. For Figure 18, if we apply pairwise joins in the order $(t_1(G) \bowtie t_2(G)) \bowtie t_3(G)$, the first join $(t_1(G) \bowtie t_2(G))$ yields 25 intermediate results, with 5 final results produced with the second join. If we rather evaluate $(t_2(G) \bowtie t_3(G)) \bowtie t_1(G)$, the first join $(t_2(G) \bowtie t_3(G))$ produces only 5 intermediate results, before the second join produces the 5 final results. The second plan should thus be more efficient than the first; if considering a graph at larger scale, the differences may reach orders of magnitude.

A good plan depends not only on the query, but also the graph. Selecting a good plan thus typically requires some assumptions or statistics over the graph. As in relational settings, the most important information relates to *cardinalities*: how many (distinct) solutions a given pattern returns; and/or *selectivity*: what percentage of solutions are kept when restricting variables with constants or filters. Statistics can be used not only to select an ordering for joins, but also to decide which join algorithm to apply. For example, given an arbitrary (sub-)BGP $\{t_1, t_2\}$, if we estimate that $|t_2(G)| \ll |t_1(G)|$, we may prefer to evaluate $t_2(G) \bowtie t_1(G)$ as an index nested-loop join, rather than a hash or merge join, to avoid reading $t_1(G)$ in full.

While cardinality and selectivity estimates can be managed in a similar way to relational database optimizers, a number of approaches have proposed custom statistics for RDF. Stocker et al. [215] collect statistics relating to the number of triples, the number of unique subjects, and for each predicate, the number of triples and a histogram of associated objects. RDF-3X [168] uses a set of *aggregated indexes*, which store the cardinality of all triple patterns with one or two constants. RDF-3X [168] further stores the exact cardinality of frequently encountered joins, while characteristic sets [165] and extended characteristic sets [155] (discussed in Section 5.3) capture the cardinality of star joins.

Computing and maintaining such statistics incur costs in terms of space and updates. An alternative is to apply *sampling* while evaluating the query. Vidal et al. [231] estimate the cardinality of star joins by evaluating all solutions for the first pattern of the join, thereafter computing the full solutions of the star pattern for a sample of the initial solutions; the full cardinality of the star pattern is then estimated from the samples. Another alternative is to use *syntactic heuristics* for reordering. Stocker et al. [215] propose heuristics such as assuming that triple patterns with fewer variables have lower cardinality, that subject constants are more selective than objects and predicates, etc. Tsialiamanis et al. [227] further propose to prioritize rarer joins (such as P–S and P–O joins), and to consider literals as more selective than IRIs.

Taking into account such heuristics and statistics, the simplest strategy to try to find a good join ordering is to ap-ply a greedy metaheuristic [215, 155], starting with the triple pattern t_1 estimated to have the lowest cardinality, and joining it with the triple pattern t_2 with the next lowest cardinality; typically a constraint is added such that t_n ($n > 1$) should have a variable in common with some triple pattern in $\{t_1, \ldots, t_{n-1}\}$ to avoid costly Cartesian products. Aside from considering the cardinality of triple patterns, Meimaris and Papastefanatos [154] propose a distance-based planning, where pairs of triple patterns with more overlapping nodes and more similar cardinality estimates have lesser distance between them; the query planner then tries to group and join triple patterns with the smallest distances first in a greedy manner. Greedy strategies will not, however, always provide the best ordering corresponding to an optimal plan.

More generally, reordering joins is an optimization problem, where classical methods from the relational literature can be leveraged likewise for BGPs, including dynamic programming [209] (used, e.g., by [94, 168, 82]) and simulated annealing [106] (used, e.g., by [231]). Other metaheuristics that have been applied for join reordering in BGPs include genetic algorithms [102] and ant colony systems [101, 114].

6.6 Caching

Another possible route for optimization – based on the observation that queries in practice may feature overlapping or similar patterns – is to reuse work done previously for other queries. Specifically, we can consider *caching* the results of queries. In order to increase cache hit rates, we can further try to reuse the results of subqueries, possibly generalizing them to increase usability. Ideally the cache should store solutions for subqueries that (a) have a high potential to reduce the cost of future queries; (b) can reduce costs for many future queries; (c) do not have a high space overhead; and (d) will remain valid for a long time. Some of these aims can be antagonistic; for example, caching solutions for triple patterns satisfies (b) and (c) but not (a), while caching solutions for complex BGPs satisfies (a) but not (b), (c) or (d).

Lampo et al. [132] propose caching of solutions for star joins, which may strike a good balance in terms of reducing costs, being reusable, and not having a high space overhead (as they share a common variable). Other caching techniques try to increase cache hit rates by detecting similar (sub)queries. Stuckenschmidt [217] uses a similarity measure for caching – based on the edit distance between BGPs – that estimates the amount of computational effort needed to compute the solutions for one query given the solutions to the other. Lorey and Naumann [140] propose a technique for grouping similar queries, which enables a pre-fetching strategy based on predicting what a user might be interested in based on their initial queries. Another direction is to normalize (sub)queries to increase cache hit rates. Wu et al. [246]

propose various algebraic normalizations in order to identify common subqueries [140], while Papailiou et al. [179] generalize subqueries by replacing selective constants with variables and thereafter canonically labeling variables (modulo isomorphism) to increase cache hit rates. Addressing dynamic data, Martin et al. [150] propose a cache where results for queries are stored in a relational database but are invalidated when a triple matching a query pattern changes. Williams and Weaver [242] add last-updated times to their RDF index to help invalidate cached data.

Given that an arbitrary BGP can produce an exponential number of results, Zhang et al. [260] propose to cache frequently accessed "hot triples" from the RDF graph in memory, rather than caching (sub-)query results. This approach limits the space overhead at the cost of recomputing joins.

6.7 Discussion

Techniques for processing BGPs are often based on techniques for processing relational joins. Beyond standard pairwise joins, multiway joins can help to emulate some of the benefits of property table storage by evaluating star joins more efficiently. Another recent and promising approach is to apply wco join algorithms whose runtime is bounded theoretically by the number of results that the BGP could generate. More and more attention has also been dedicated to computing joins in GPUs by translating relational algebra (e.g., joins) into linear algebra (e.g., matrix multiplication). Aside from specific algorithms, the order in which joins are processed can have a dramatic effect on runtimes. Statistics about the RDF graph help to find a good ordering at the cost of computing and maintaining those statistics; more lightweight alternatives include runtime sampling, or syntactic heuristics that consider only the query. To decide the ordering, options range from simple greedy strategies to complex metaheuristics; while simpler strategies have lower planning times, more complex strategies may find more efficient plans. Another optimization is to cache results across BGPs, for which a time–space trade-off must be considered.

7 Query Processing

While we have defined RDF stores as engines capable of storing, indexing and processing joins over RDF graphs, SPARQL engines support various features beyond joins. We describe techniques for efficiently evaluating such features, including the relational algebra (beyond joins) and property paths. We further include some general extensions proposed for SPARQL to support recursion and analytics.

7.1 Relational algebra (beyond joins)

Complex (navigational) graph patterns CGPs introduce additional relational operators beyond joins.

Like in relational databases, *algebraic rewriting rules* can be applied over CGPs in SPARQL to derive equivalent but more efficient plans. Schmidt et al. [207] present a set of such rules for SPARQL under set semantics, such as:

$$\sigma_{R_1 \vee R_2}(M) \equiv \sigma_{R_1}(M) \cup \sigma_{R_2}(M)$$
$$\sigma_{R_1 \wedge R_2}(M) \equiv \sigma_{R_1}(\sigma_{R_2}(M))$$
$$\sigma_{R_1}(\sigma_{R_2}(M)) \equiv \sigma_{R_2}(\sigma_{R_1}(M))$$
$$\sigma_R(M_1 \cup M_2) \equiv \sigma_R(M_1) \cup \sigma_R(M_2)$$
$$\sigma_R(M_1^* \bowtie M_2) \equiv \sigma_R(M_1^*) \bowtie M_2$$
$$\sigma_R(M_1^* \rhd\!\!\bowtie M_2) \equiv \sigma_R(M_1^*) \rhd\!\!\bowtie M_2$$
$$\sigma_R(M_1^* \rhd M_2) \equiv \sigma_R(M_1^*) \rhd M_2$$

where for each $\mu \in M_1^*$, it holds that $\text{vars}(R) \subseteq \text{dm}(\mu)$. The first two rules split filters, meaning that they can be pushed further down in a query in order to reduce intermediary results. The third rule allows the order in which filters are applied to be swapped. Finally the latter four rules describe how filters can be pushed "down" inside various operators.

Another feature of importance for querying RDF graphs are optionals ($\rhd\!\!\bowtie$), as they facilitate returning partial solutions over incomplete data. Given that an optional can be used to emulate a form of negation (in Table 3 it is defined using an anti-join), it can lead to jumps in computational complexity [183]. Works have thus studied a fragment called *well-designed patterns*, which forbid using a variable on the right of an optional that does not appear on the left but does appear elsewhere in the query; taking an example, the CGP $(\{(\mathbf{x}, p, \mathbf{y})\} \ \text{OPTIONAL} \ \{(\mathbf{x}, q, \mathbf{z})\}) \ . \ \{(\mathbf{x}, r, \mathbf{z})\}$ is not well designed as the variable $\mathbf{z}$ appears on the right of an OPTIONAL and not on the left, but does appear elsewhere in the query. Such variables may or may not be left unbound after the left outer join is evaluated, which leads to complications if they are used outside the optional clause. Most SPARQL queries using optionals in practice are indeed well-designed, where rewriting rules have been proposed specifically to optimize such queries [183, 137].

7.2 Property paths

Navigational graph patterns (NGPs) extend BGPs with property paths, which are extensions of (2)RPQs that allow for matching paths of arbitrary length in the graph.

Some approaches evaluate property paths using graph search algorithms. Though not part of SPARQL, Gubichev and Neumann [81] implement single-source shortest paths by applying Dijkstra's search algorithm over B-Trees. Baier et al. [23] propose to use the A* search algorithm, where

search is guided by a heuristic that measures the minimum distance from the current node to completing a path.

Extending RDF-3X, Gubichev et al. [80] build a FER-RARI index [210] (see Section 5.4) for each property :p in the graph that forms a directed path of length at least 2. The indexes are used to evaluate paths :p* or :p+. Paths of the form (:p/:q)*, (:p|:q)*, etc., are not directly supported.

Koschmieder and Leser [127], and Nguyen and Kim [170] optimize property paths by splitting them according to "rare labels": given a property path :p * /:q/:r*, if :q has few triples in the graph, the path can be split into :p*/:q (evaluated right-to-left) and :q/:r* (evaluated left-to-right), subsequently joining the results. Splitting paths can enable parallelism: Miura et al. [158] evaluate such splits on field programmable gate arrays (FPGAs), enabling hardware acceleration. Wadhwa et al. [234] rather use bidirectional random walks from candidate endpoints on both sides of the path, returning solutions when walks from each side coincide.

Another way to support property paths is to use recursive queries. Stuckenschmidt et al. [218] evaluate property paths such as :p+ using recursive nested-loop and hash joins. Dey et al. [63], Yakovets et al. [252] and Jachiet et al. [108] propose translations of more general property paths (or RPQs) to extensions of the relational algebra with recursive or transitive operators. Paths can be evaluated by SQL engines using WITH RECURSIVE; however Yakovets et al. [252] note that highly nested SQL queries may result, and that popular relational database engines cannot (efficiently) detect cycles. Dey et al [63] alternatively explore the evaluation of RPQs via translations to recursive Datalog.

In later work, Yakovets et al. [253] propose Waveguide, which first converts the property path into a parse tree, from which plans can be built based on finite automata (FA), or relational algebra with transitive closure (α-RA, where α denotes transitive closure). Figure 19 gives an example of a parse tree and both types of plans. Although there is overlap, FA can express physical plans that α-RA cannot, and vice versa. For example, in FA we can express non-deterministic transitions (see q_0 in Figure 19), while in α-RA we can materialize (cache) a particular relation in order to apply transitive closure over it. Waveguide then uses hybrid *waveplans*, where breadth-first search is guided in a similar manner to FA, but where the results of an FA can be *memoized* (cached) and reused multiple times like in α-RA.

Evaluating complex property paths can be costly, but property paths in practice are often quite simple. Martens and Trautner [149] propose a class of RPQs called *simple transitive expressions* (STEs) that are found to cover 99.99% of the queries found in Wikidata SPARQL logs, and have desirable theoretical properties. Specifically, they define *atomic expressions* of the form $p_1 | \ldots | p_n$, where $p_1, \ldots, p_n$ are IRIs and $n \geq 0$; and also *bounded expressions* of the form $a_1 / \ldots / a_k$ or $a_1? / \ldots / a_k?$ where $a_1, \ldots, a_k$ are atomic

expressions and $k \geq 0$. Then an expression of the form $b_1 / a^* / b_2$, is a *simple transitive expression* (*STE*), where b_1 and b_2 are bounded expressions, and a is an atomic expression. They then show that simple paths for STEs can be enumerated more efficiently than arbitrary RPQs.

7.3 Recursion

Property paths offer a limited form of recursion. While extended forms of property paths have been proposed to include (for example) path intersection and difference [71], more general extensions of SPARQL have also been proposed to support graph-based and relation-based recursion.

Reutter et al. [194] propose to extend SPARQL with graph-based recursion, where a temporary RDF graph is built by recursively adding triples produced through CONSTRUCT queries over the base graph and the temporary graph up to a fixpoint; a SELECT query can then be evaluated over both graphs. The authors discuss how key features (including property paths) can then be supported through *linear recursion*, meaning that each new triple only needs to be joined with the base graph, not the temporary graph, to produce further triples, leading to better performance. Corby et al. [59] propose LD-Script: a SPARQL-based scripting language supporting various features, including for-loops that can iterate over the triples returned by a CONSTRUCT query.

Hogan et al. [98] propose SPARQAL: a lightweight language that supports relation-based (i.e., SELECT-based) recursion over SPARQL. The results of a SELECT query can be stored as a variable, and injected into a future query. Do–until loops can be called until a particular condition is met, thus enabling recursion over SELECT queries.

7.4 Analytics

SPARQL engines often focus on transactional (OLTP) workloads involving selective queries that are efficiently solved through lookups on indexes. Recently, however, a number of approaches have looked at addressing analytical (OLAP) workloads for computing slices, aggregations, etc. [45].

One approach is to rewrite SPARQL queries to languages executable in processing environments suitable for analytical workloads, including PigLatin (e.g., PigSPARQL [202], RAPID+ [193]), Hadoop (e.g., Sempala [203]), Spark (e.g., S2RDF [204]), etc. Such frameworks are better able to handle analytical (OLAP) workloads, but not all SPARQL features are easily supported on existing distributed frameworks.

Conversely, one can also translate from analytical languages to SPARQL queries, allowing for *in-database analytics*, where analytical workloads are translated into queries run by the SPARQL engine/database. Papadaki et al. [176] propose the high-level functional query language HIFUN for
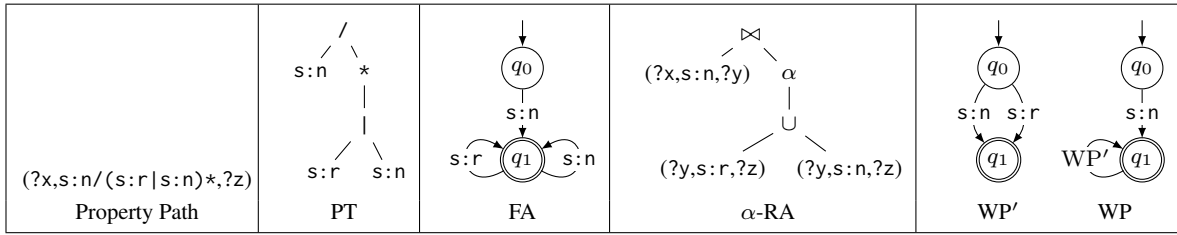
Fig. 19: An example property path with its parse tree (PT) and three plans based on finite automata (FA), relational algebra with transitive closure (α-RA), and a waveplan (WP) that uses a memoized waveplan (WP$'$)

applying analytics over RDF data. Rules for translating analytical HIFUN queries to SPARQL are then presented.

There has also been growing interest in combining *graph analytics* – such as centrality measures, shortest paths, graph clustering, etc. – with SPARQL. In this way, SPARQL can be used as a declarative language to construct sub-graphs over which analytics are applied, and can further express queries involving the results of analytics. Unlike OLAP-style analytics, graph analytics often require recursion. One approach is to extend SPARQL to include imperative functions for invoking common graph algorithms. Abdelaziz et al. [4] propose Spartex: an extension of SPARQL that allows for invoking common graph algorithms – such as PageRank, shortest paths, etc. – as well as user-defined procedures (UDPs) written in a custom procedural language. An alternative approach is to support graph analytics through a more general recursive language based on SPARQL (as discussed in Section 7.3). Hogan et al. [98] show how the recursive language SPARQAL allows for expressing and evaluating in-database graph analytics, including breadth-first search, PageRank, local clustering coefficient, etc.

7.5 Graph query rewriting

We have seen approaches that rewrite SPARQL queries into languages such as SQL [69, 252], PigLatin [202, 193], etc. Other works rewrite SPARQL into the query languages of (other) graph databases. SPARQL–Gremlin [223] rewrites SPARQL to Gremlin, allowing SPARQL queries to be evaluated on graph database engines that support Gremlin, while Semantic Property Graph [190] describes how reified RDF graphs can be projected into the property graph model supported by many graph database engines.

7.6 Multi-query optimization

While the techniques discussed thus far optimize queries individually, multi-query optimization evaluates batches of queries efficiently by exploiting their commonalities. Le et al. [133] propose to first cluster a set of queries into groups with maximal common edge subgraphs; for example, the

BGP $\{(\mathbf{w_1}, p, \mathbf{x_1}), (\mathbf{w_1}, q, \mathbf{y_1}), (\mathbf{w_1}, r, \mathbf{z_1}), (\mathbf{y_1}, s, \mathbf{z_1})\}$ and the BGP $\{(\mathbf{w_2}, p, \mathbf{x_2}), (\mathbf{w_2}, q, \mathbf{y_2}), (\mathbf{z_2}, r, \mathbf{w_2})\}$ may form a cluster. A query is then constructed for each cluster by extending its maximal common sub-BGP with optional patterns needed by a proper subset of the queries; for example, $(\{(\mathbf{w}, p, \mathbf{x}), (\mathbf{w}, q, \mathbf{y})\} \bowtie \{(\mathbf{w}, r, \mathbf{z}), (\mathbf{y}, s, \mathbf{z})\}) \bowtie \{(\mathbf{z}, r, \mathbf{w})\}$ would be used for the previous cluster. Individual query results are then computed from the cluster-level results. Optimizing for multiple property paths, Abul-Basher [7] proposes to find a maximum common sub-automaton that can be evaluated and reused across multiple queries. More recent works further address multi-query optimization in specific settings, including federated systems [180], and continuous querying over streaming RDF data [259].

7.7 Discussion

SPARQL supports various features beyond joins that ideally should be implemented in an efficient manner. One option is to rewrite SPARQL queries into a target language and evaluate them using an existing engine for that language. However, it is unlikely that an existing language/engine will support all features of SPARQL in an efficient manner. Better performance for a wider range of features can be achieved with custom implementations and optimizations, where property paths have been the focus of many works. Other features that have been targeted for optimization are filters and optionals, noting that optionals are quite frequently used in the context of querying incomplete RDF data. Multi-query optimization can further help to evaluate multiple queries at once. More recent works have addressed recursion and analytics for SPARQL in order to support additional RDF data management scenarios and knowledge graph use-cases.

8 Partitioning

In distributed RDF stores and SPARQL engines, the data are partitioned over a cluster of machines in order to enable horizontal scale, where additional machines can be allocated to the cluster to handle larger volumes of data. However, horizontal scaling comes at the cost of network communication

costs. Thus a key optimization is to choose a partitioning scheme that reduces communication costs by enforcing various forms of locality, principally allowing certain types of (intermediate) joins to be processed on each individual machine [8]. Formally, given an RDF graph G and n machines, an n-partition of G is a tuple of subgraphs $(G_1, \ldots, G_n)$ such that $G = \bigcup\limits_{i=1}^{n} G_i$, with the idea that each subgraph G_i will be stored on machine i.[5] We now discuss different high-level alternatives for partitioning.

8.1 Triple/Quad-based Partitioning

A first option is to partition based on individual triples or quads without considering the rest of the graph. For simplicity we will speak about triples as the discussion generalizes straightforwardly to quads. The simplest option is to use *round robin* or *random partitioning*, which effectively places triples on an arbitrary machine. This ensures even load balancing, but does not support any locality of processing, and does not allow for finding the particular machine storing triples that match a given pattern.

An alternative is to partition according to a deterministic function over a given key; for example, a partition key of s considers only the subject, while a partition key of po considers both the predicate and object. Later given a triple pattern that covers the partition key (e.g., with a constant subject if the key is s), we can find the machine(s) storing all triples that match that pattern. We show some examples using different functions and partition keys in Figure 20 considering four machines. *Range-based partitioning* assigns a range over the partition key to each function, where the example of Figure 20 splits s into [a:1,a:3], [a:4,a:6], [b:1,b:3], [c:1,d:1]. This approach allows for range-based queries to be pushed to one machine, but requires maintaining a mapping of ranges to machines, and can be complicated to keep balanced. An alternative is *hash-based partitioning* where we compute the hash of the partition key modulo the number of machines, where the second example of Figure 20 splits p by hash. This does not require storing any mapping, and techniques such as consistent hashing can be used to rebalance load when a machine enters or leaves; however, if partition keys are skewed (e.g., one predicate is very common), it may lead to an unbalanced partition. A third option is to apply a *hierarchical-based partition* based on prefixes, where the third example of Figure 20 partitions o by their namespace. This may lead to increased locality of data with the same prefix [113], where different levels of prefix can be chosen to enable balancing, but choosing prefixes that offer balanced partitions is non-trivial.

Any such partitioning function will send any triple with the same partition key to the same machine, which ensures that (equi-)joins on partition keys can be pushed to individual machines. Hash-based partitioning is perhaps the most popular among distributed RDF stores (e.g., YARS2 [94], SHARD [195], etc.). Often triples will be hashed according to multiple partition keys in order to support different index permutations, triple patterns, and joins (e.g, with s and o as two partition keys, we can push s–s, o–o and s–o joins to each machine). Care must be taken to avoid imbalances caused by frequent terms, such as the rdf:type predicate, or frequent objects such as classes, countries, etc. Omitting partitioning on highly-skewed partition keys may be advantageous for balancing purposes [94].

8.2 Graph-based Partitioning

Graph-based partitioning takes into consideration the entire graph when computing a partition. A common strategy is to apply a *k-way partition* of the RDF graph G [119]. Formally, letting $V = \text{so}(G)$ denote the nodes of G, the goal is to compute a node partition $V_1, \ldots, V_n$ such that $V = \bigcup\limits_{i=1}^{k} V_n$, $V_i \cap V_j = \emptyset$ for all $1 \le i < j \le k$, $\lfloor \frac{|V|}{k} \rfloor \le |V_i| \le \lceil \frac{|V|}{k} \rceil$ for all $1 \le i \le k$, and the number of triples $(s, p, o) \in G$ such that s and o are in different node partitions is minimized. In Figure 21, we show the optimal 4-way partitioning of the graph seen previously, where each partition has 3 nodes, there are 10 edges between partitions (shown dashed), and no other such partition leads to fewer edges (<10) between partitions. Edges between partitions may be replicated in the partitions they connect. Another alternative is to k-way partition the *line graph* of the RDF graph: an undirected graph where each triple is a node, and triples sharing a subject or object have an edge between them.

Finding an optimal k-way partition is intractable[6], where approximations are thus necessary for large-scale graphs, including *spectral methods*, which use the eigenvectors of the graph's Laplacian matrix to partition it; *recursive bisection*, which recursively partitions the graph into two; *multilevel partitioning*, which "coarsens" the graph by computes a hierarchical graph summary (similar to a multilevel quotient graph, per Figure 17), then partitions the smaller graph summary (using, e.g., spectral methods), and finally "uncoarsens" by expanding back out to the original graph maintaining the partitions; etc. We refer for more details to Buluç et al. [44], who argue that multilevel partitioning is "*clearly the most successful heuristic for partitioning large graphs*". Such techniques have been used by H-RDF-3x [104], EAGRE [261], Koral [112], and more besides.

---

[5] We relax the typical requirement for a set partition that $G_i \cap G_j = \emptyset$ for all $1 \le i < j \le n$ to allow for the possibility of replication or other forms of redundancy.

[6] Given a graph, deciding if there is a k-way partition with fewer than n edges between partitions is NP-complete.

Fig. 20: Examples of triple-based partitioning schemes

8.3 Query-based Partitioning

While the previous partitioning schemes only consider the data, other partitioning methods are (also) based on queries. Workload-based partitioning schemes identify common joins in query logs that can be used to partition or replicate parts of the graph in order to ensure that high-demand joins can be pushed to individual machines. Partitioning can then be *a priori*, for example, based on a query log; or *dynamic* (aka. adaptive), where the partitions change as queries are received. Such strategies are used by systems that include WARP [103], Partout [73], WORQ [146], and AdPart [89].

8.4 Replication

Rather than partitioning data, data can also be replicated across partitions. This may vary from replicating the full graph on each machine, such that queries can be answered in full by any machine to increase query throughput (used, e.g., by DREAM [88]), to replicating partitions that are in high-demand (e.g., containing schema data, central nodes, etc.) so that more queries can be evaluated on individual machines and/or machines have equal workloads that avoid hot-spots (used, e.g., by Blazegraph [224] and Virtuoso [69]).

8.5 Discussion

Triple/quad-based partitioning is the simplest to compute and maintain, being dependent only on the data present in an individual tuple, allowing joins on the same partition key to be pushed to individual machines. Graph-based partitions allow for evaluating more complex graph patterns on individual machines, but are more costly to compute and maintain (considering, e.g., dynamic data). Information about queries, where available, can be used for the purposes of workload-based partitioning, which partitions or replicates data in or-

Fig. 21: Example of optimal k-way partitioning ($k = 4$)

der to enable locality for common sub-patterns. Replication can further improve load balancing, locality and fault-tolerance at the cost of redundant storage.

9 Systems and Benchmarks

In Appendix A we present a comprehensive survey of 135 individual RDF stores and SPARQL query engines – both distributed and local – in terms of the techniques discussed herein that they use. In Appendix B, we further present the synthetic and real-world benchmarks available for evaluating these systems under a variety of criteria.

10 Summary

In order to conclude this survey paper, we first summarize some of the current high-level trends that we have observed while preparing this survey, and then summarize the open research challenges that are left to address.

10.1 Current trends

While RDF stores and SPARQL engines have traditionally relied on relational databases and relational-style optimizations to ensure scalability and efficiency, we see a growing trend towards (1) native graph-based storage, indexing and query processing techniques, along with (2) exploiting modern hardware and data management/processing.

Native storage techniques for graphs move away from relational-style schemata for RDF, and rather focus on optimizing for the compression and navigation of RDF as a graph, with techniques such as index-free adjacency, tensor-based storage, and other graph-based representations. Indexing likewise has evolved to consider entity-based (i.e., node-based) schemes, path indexes, and structural indexes based on summarizing the graph structure of RDF data. While join processing over RDF is still largely inspired by techniques for relational databases, algorithms based on sideways information passing, multi-way joins, worst-case optimal joins, etc., have been shown to work particularly well on RDF graphs (e.g., given their fixed arity). In terms of query processing, features such as property paths and graph-based recursion go beyond what is considered in typical relational database management, with increased attention being paid to supporting graph analytics in the RDF/SPARQL setting.

Regarding modern hardware, following broader trends, many works now leverage NoSQL systems and distributed processing frameworks in order to scale RDF stores across multiple machines and handle new types of workloads. A similar trend is to better exploit modern hardware, where a variety of compact data structures have been proposed for storing RDF graphs in main memory, possibly across multiple machines, following a general trend of exploiting the growing RAM capacity of modern hardware. Recent techniques for processing graphs – represented as matrices/tensors – further enable hardware acceleration by leveraging GPUs and HPC architectures, per machine learning.

Such trends seem set to continue, where we expect to see further proposals of "native" techniques for RDF/SPARQL, further works that bridge from the RDF/SPARQL setting to related data management and processing settings in order to better support other types of workloads, as well as techniques that better leverage modern hardware, including increased RAM capacity, solid-state disks, GPUs, clusters of machines, and HPC architectures.

10.2 Research Challenges and Future Directions

Though major advances have been made in terms of the *scale* and *efficiency* of RDF stores in recent years, these will remain central challenges as the scale of RDF graphs and demand for querying them in more complex ways increases.

Other challenges have only been occasionally or partially addressed by the literature, where we highlight:

Dynamics: Many of the surveyed works assume static data, and do not handle updates gracefully. Thus, more work is needed on efficiently querying dynamic RDF graphs with SPARQL, including storage that efficiently supports reads and writes, incremental indexing, caching, etc.

Query optimizations (beyond joins): Most works focus on optimizing joins and basic graph patterns. We found relatively few works optimizing features of SPARQL 1.1, such as property paths, negation, etc., where more work is needed. The expressivity of the SPARQL language is sure to grow (e.g., in the context of SPARQL 1.2), where these new features will likewise call for new techniques.

Query volume: Leading SPARQL endpoints process millions of queries per day. This challenge motivates further research on workload-aware or caching strategies that leverage frequent sub-queries. Another research challenge is on how to ensure effective policies for serving many clients while avoiding server overload, where methods such as *pre-emption* [157], which allows for pausing and resuming costly query requests, are promising ideas for further development.

Evaluation: Various benchmarks are now available for comparing different RDF stores, but they tend to focus on system-level comparisons, thus conflating techniques. More fine-grained evaluation at the level of individual techniques in the RDF/SPARQL setting would be very useful to understand the different trade-offs that exist. Also many benchmarks were proposed for SPARQL 1.0, where there is a lack of benchmarks including features such as property paths.

Integration: RDF and SPARQL are widely adopted on the Web, and for managing and querying knowledge graphs. However, in such settings, additional types of tasks are often considered, including *federated querying*, *reasoning*, *enrichment*, *refinement*, *learning*, *analytics*, etc. More work is needed on supporting or integrating features for these tasks in SPARQL. Interesting questions relate to efficiently supporting RDFS/OWL/Datalog reasoning, graph algorithms, knowledge graph embeddings, graph neural networks, etc., for RDF graphs within SPARQL engines.

References

1. D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.*, 18(2):385–406, 2009.

2. D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, pages 411–422. ACM, 2007.

3. I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *PVLDB*, 10(13):2049–2060, 2017.

4. I. Abdelaziz, R. Harbi, S. Salihoglu, and P. Kalnis. Combining Vertex-Centric Graph Processing with SPARQL for Large-Scale RDF Data Analytics. *IEEE TPDS*, 28(12):3374–3388, 2017.

5. I. Abdelaziz, R. Harbi, S. Salihoglu, and P. Kalnis. Combining Vertex-Centric Graph Processing with SPARQL for Large-Scale RDF Data Analytics. *IEEE Trans. Parallel Distributed Syst.*, 28(12):3374–3388, 2017.

6. F. Abiri, M. Kahani, and F. Zarinkalam. An entity based RDF indexing schema using Hadoop and HBase. In *International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 68–73, 2014.

7. Z. Abul-Basher. Multiple-Query Optimization of Regular Path Queries. In *International Conference on Data Engineering (ICDE)*, pages 1426–1430. IEEE, 2017.

8. A. Akhter, A.-C. N. Ngonga, and M. Saleem. An empirical evaluation of RDF graph partitioning techniques. In *European Knowledge Acquisition Workshop*, pages 3–18. Springer, 2018.

9. K. Alaoui. A Categorization of RDF Triplestores. In *International Conference on Smart City Applications (SCA)*, pages 1–7. ACM, 2019.

10. W. Ali, M. Saleem, B. Yao, A. Hogan, and A. N. Ngomo. A Survey of RDF Stores & SPARQL Engines for Querying Knowledge Graphs. *CoRR*, abs/2102.13027, 2020. `https://arxiv.org/abs/2102.13027`.

11. G. Aluç, M. T. Özsu, K. Daudjee, and O. Hartig. chameleon-db: a Workload-Aware Robust RDF Data Management System, 2013.

12. G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *International Semantic Web Conference (ISWC)*, pages 197–212. Springer, 2014.

13. S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, M. A. Martínez-Prieto, and G. Navarro. Compressed vertical partitioning for efficient RDF management. *Knowl. Inf. Syst.*, 44(2):439–474, 2015.

14. J. Anderson. RDF Graph Stores as Convergent Datatypes. In *Companion of The 2019 World Wide Web Conference*, pages 940–942. ACM, 2019.

15. J. Anderson and A. Bendiken. Transaction-Time Queries in Dydra. In *Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW@ESWC)*, volume 1585, pages 11–19. CEUR-WS.org, 2016.

16. R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of Modern Query Languages for Graph Databases. *ACM CSUR*, 50(5):68:1–68:40, 2017.

17. A. Aranda-Andújar, F. Bugiotti, J. Camacho-Rodríguez, D. Colazzo, F. Goasdoué, Z. Kaoudi, and I. Manolescu. AMADA: Web Data Repositories in the Amazon Cloud. In *Conference on Information and Knowledge Management (CIKM)*, page 2749–2751. ACM, 2012.

18. N. Arndt. Tripleplace: A flexible triple store for android with six indices. *Studentenkonferenz Informatik Leipzig*, pages 1–8, 2011.

19. D. Arroyuelo, A. Hogan, G. Navarro, J. L. Reutter, J. Rojas-Ledesma, and A. Soto. Worst-Case Optimal Graph Joins in Almost No Space. In *SIGMOD International Conference on Management of Data*, pages 102–114. ACM, 2021.

20. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In *International Conference on World Wide Web (WWW)*, pages 41–50. ACM, 2010.

21. M. Atre and J. A. Hendler. BitMat: A Main Memory Bit-matrix of RDF Triples. In *Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, page 33, 2009.

22. A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

23. J. Baier, D. Daroch, J. L. Reutter, and D. Vrgoc. Evaluating Navigational RDF Queries over the Web. In *ACM Conference on Hypertext and Social Media (HT)*, pages 165–174. ACM, 2017.

24. S. Bail, S. Alkiviadous, B. Parsia, D. Workman, M. van Harmelen, R. S. Gonçalves, and C. Garilao. FishMark: A Linked Data Application Benchmark. In *Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+HPCSW)*, pages 1–15. CEUR, 2012.

25. M. Banane. RDFMongo: A MongoDB Distributed and Scalable RDF management system based on Meta-model. *Int. J. of Adv. Trends in Comp. Sci. and Eng.*, 8:734–741, 2019.

26. S. Bartoň. Designing Indexing Structure for Discovering Relationships in RDF Graphs. In *International Workshop on DAtabases, TExts, Specifications and Objects (DATESO)*, pages 7–17. CEUR, 2004.

27. B. R. Bebee, D. Choi, A. Gupta, A. Gutmans, A. Khandelwal, Y. Kiran, S. Mallidi, B. McGaughy, M. Personick, K. Rajan, S. Rondelli, A. Ryazanov, M. Schmidt, K. Sengupta, B. B. Thompson, D. Vaidya, and S. Wang. Amazon Neptune: Graph Data Management in the Cloud. In *ISWC Posters & Demonstrations, Industry and Blue Sky Ideas Tracks*. CEUR-WS.org, 2018.

28. D. Beckett. The design and implementation of the redland RDF application framework. In *World Wide Web Conference (WWW)*, pages 449–456. ACM, 2001.

29. C. Beeri and R. Ramakrishnan. On the Power of Magic. *J. Log. Program.*, 10(3&4):255–299, 1991.

30. D. Bennett, J. Engelbrecht, and D. Landau. CM-Well: A Data Warehouse for Linked Data. In *ISWC Posters & Demonstrations and Industry Tracks*, volume 1963. CEUR-WS.org, 2017.

31. A. Bigerl, F. Conrads, C. Behning, M. Sherif, M. Saleem, and A.-C. N. Ngomo. Tentris: A Tensor-Based Triple Store. In *International Semantic Web Conference (ISWC)*, pages 56–73. Springer, 2020.

32. R. Binna, W. Gassler, E. Zangerle, D. Pacher, and G. Specht. SpiderStore: A Native Main Memory Approach for Graph Storage. In *Grundlagen von Datenbanken (GI-Workshop)*, pages 91–96. CEUR, 2011.

33. B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. OWLIM: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011.

34. C. Bizer, R. Meusel, and A. Primpel. Web Data Commons – Microdata, RDFa, JSON-LD, and Microformat Data Sets, 2020. `http://webdatacommons.org/structureddata/`.

35. C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *Int. J. Sem. Web Inf. Syst.*, 5(2):1–24, 2009.

36. M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an Efficient RDF Store over a Relational Database. In *International Conference on Management of Data (SIGMOD)*, page 121–132. ACM, 2013.

37. D. Brickley and R. Guha. RDF Schema 1.1. W3C Recommendation, 2014. https://www.w3.org/TR/rdf-schema/.

38. N. R. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, and A. Fariña. Revisiting Compact RDF Stores Based on k2-Trees. In *Data Compression Conference (DCC)*, pages 123–132. IEEE, 2020.

39. N. R. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, A. Fariña, and G. Navarro. Space/time-efficient RDF stores based on circular suffix sorting. *CoRR*, abs/2009.10045, 2020.

40. N. R. Brisaboa, A. Cerdeira-Pena, A. Fariña, and G. Navarro. A Compact RDF Store Using Suffix Arrays. In *String Processing and Information Retrieval (SPIRE)*, pages 103–115. Springer, 2015.

41. M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. DOGMA: A Disk-Oriented Graph Matching Algorithm for RDF Databases. In *International Semantic Web Conference (ISWC)*, pages 97–113. Springer, 2009.

42. J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *International Semantic Web Conference (ISWC))*, pages 54–68. Springer, 2002.

43. C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *International Semantic Web Conference (ISWC)*, pages 277–293. Springer, 2013.

44. A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

45. C. Callé, O. Cure, and P. Calvez. Motivations for an Analytical RDF Database System, 2021. https://openreview.net/forum?id=M4H2AdgOhFX.

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

47. P. Cappellari, R. D. Virgilio, and M. Roantree. Path-oriented keyword search over graph-modeled Web data. *World Wide Web Conference (WWW)*, 15(5-6):631–661, 2012.

48. S. Cebiric, F. Goasdoué, H. Kondylakis, D. Kotzinos, I. Manolescu, G. Troullinou, and M. Zneika. Summarizing semantic graphs: a survey. *VLDBJ*, 28(3):295–327, 2019.

49. C. Chantrapornchai and C. Choksuchat. TripleID-Q: RDF Query Processing Framework Using GPU. *IEEE TPDS*, 29(9):2121–2135, 2018.

50. T. Chawla, G. Singh, E. Pilli, and M. Govil. Storage, partitioning, indexing and retrieval in Big RDF frameworks: A survey. *Computer Science Review*, 38:100309, 11 2020.

51. X. Chen, H. Chen, N. Zhang, and S. Zhang. SparkRDF: Elastic Discreted RDF Graph Processing Engine with Distributed Memory. In *ISWC Posters & Demos (ISWC-PD)*, pages 261–264. CEUR, 2014.

52. Y. Chen, M. T. Özsu, G. Xiao, Z. Tang, and K. Li. GSmart: An Efficient SPARQL Query Engine Using Sparse Matrix Algebra – Full Version, 2021.

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

54. S. K. Chirravuri. RDF3X-MPI: A partitioned RDF engine for data-parallel SPARQL querying. Master's thesis, Pennsylvania State University, 2014.

55. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An Efficient SQL-Based RDF Querying Scheme. In *International Conference on Very Large Databases (VLDB)*, page 1216–1227. VLDB End., 2005.

56. F. Conrads, J. Lehmann, M. Saleem, M. Morsey, and A. N. Ngomo. IGUANA: A Generic Framework for Benchmarking the Read-Write Performance of Triple Stores. In *International Semantic Web Conference (ISWC)*, pages 48–65. Springer, 2017.

57. O. Corby, R. Dieng-Kuntz, and C. Faron-Zucker. Querying the semantic web with corese search engine. In R. L. de Mántaras and L. Saitta, editors, *European Conference on Artificial Intelligence (ECAI)*, pages 705–709. IOS Press, 2004.

58. O. Corby and C. Faron-Zucker. Implementation of SPARQL Query Language Based on Graph Homomorphism. In *International Conference on Conceptual Structures (ICCS)*, pages 472–475. Springer, 2007.

59. O. Corby, C. Faron-Zucker, and F. Gandon. LDScript: A Linked Data Script Language. In *International Semantic Web Conference (ISWC)*, volume 10587 of *LNCS*, pages 208–224. Springer, 2017.

60. M. Cossu, M. Färber, and G. Lausen. PRoST: Distributed Execution of SPARQL Queries Using Mixed Partitioning Strategies. In *International Conference on Extending Database Technology (EDBT)*, pages 469–472. OpenProceedings, 2018.

61. O. Curé, G. Blin, D. Revuz, and D. C. Faye. Waterfowl: A compact, self-indexed and inference-enabled immutable RDF store. In *Extended Semantic Web Conference (ESWC)*, pages 302–316. Springer, 2014.

62. G. Demartini, I. Enchev, M. Wylot, J. Gapany, and P. Cudré-Mauroux. BowlognaBench - Benchmarking RDF Analytics. In *Symposium on Data-Driven Process Discovery and Analysis (SIMPDA)*, pages 82–102. Springer, 2011.

63. S. C. Dey, V. Cuevas-Vicenttín, S. Köhler, E. Gribkoff, M. Wang, and B. Ludäscher. On implementing provenance-aware regular path queries with relational query engines. In *Joint 2013 EDBT/ICDT Conferences*, pages 214–223. ACM, 2013.

64. F. Du, H. Bian, Y. Chen, and X. Du. Efficient SPARQL Query Evaluation in a Database Cluster. In *BigData Congress*, pages 165–172. IEEE, 2013.

65. S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: A comparison of RDF benchmarks and real RDF datasets. In *International Conference on Management of Data (SIGMOD)*, pages 145–156. ACM, 2011.

66. M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). RFC 3987.

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

68. O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat-Pérez, M. Pham, and P. A. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *International Conference on Data Management (SIGMOD)*, pages 619–630. ACM, 2015.

69. O. Erling and I. Mikhailov. *Virtuoso: RDF Support in a Native RDBMS*, pages 501–519. Springer, 2010.

70. D. C. Faye, O. Curé, and G. Blin. A survey of RDF storage approaches. In *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, page pp. 25, 2012.

71. V. Fionda, G. Pirrò, and M. P. Consens. Querying knowledge graphs with extended property paths. *Semantic Web*, 10(6):1127–1168, 2019.

72. G. H. L. Fletcher, J. Peters, and A. Poulovassilis. Efficient regular path query evaluation using path indexes. In *Extending Database Technology (EDBT)*, pages 636–639. OpenProceedings.org, 2016.

73. L. Galárraga, K. Hose, and R. Schenkel. Partout: A distributed engine for efficient RDF processing. In *WWW Companion*, pages 267–268. ACM, 2014.

74. M. Galkin, K. M. Endris, M. Acosta, D. Collarana, M. Vidal, and S. Auer. SMJoin: A Multi-way Join Operator for SPARQL Queries. In *International Conference on Semantic Systems (SEMANTICS)*, pages 104–111. ACM, 2017.

75. F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis. CliqueSquare: Flat plans for massively parallel RDF queries. In *International Conference on Data Engineering (ICDE)*, pages 771–782. IEEE, 2015.

76. D. Graux, L. Jachiet, P. Genevès, and N. Layaïda. SPARQLGX: Efficient distributed evaluation of SPARQL with Apache Spark. In *International Semantic Web Conference (ISWC)*, pages 80–87. Springer, 2016.

77. J. Groppe, S. Groppe, A. Schleifer, and V. Linnemann. Lupos-Date: a semantic web database system. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 2083–2084. ACM, 2009.

78. S. Groppe, J. Groppe, and V. Linnemann. Using an index of precomputed joins in order to speed up SPARQL processing. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 13–20, 2007.

79. S. Groppe, T. Kiencke, S. Werner, D. Heinrich, M. Stelzner, and L. Gruenwald. P-LUPOSDATE: Using Precomputed Bloom Filters to Speed Up SPARQL Processing in the Cloud. *Open J. Semantic Web*, 1(2):25–55, 2014.

80. A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling kleene: fast property paths in RDF-3X. In *Workshop on Graph Data Management Experiences & Systems (GRADES)*, page 14. CWI/ACM, 2013.

81. A. Gubichev and T. Neumann. Path Query Processing on Very Large RDF Graphs. In *International Workshop on the Web and Databases (WebDB)*, 2011.

82. A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *International Conference on Extending Database Technology (EDBT)*, pages 439–450. OpenProceedings.org, 2014.

83. X. Guo, H. Gao, and Z. Zou. Leon: A Distributed RDF Engine for Multi-query Processing. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 742–759. Springer, 2019.

84. X. Guo, H. Gao, and Z. Zou. WISE: Workload-Aware Partitioning for RDF Systems. *Big Data Research*, 22:100161, 2020.

85. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

86. S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In *International Conference on Management of Data (SIGMOD)*, pages 289–300. ACM, 2014.

87. P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A Comparison of RDF Query Languages. In *International Semantic Web Conference (ISWC)*, pages 502–517. Springer, 2004.

88. M. Hammoud, D. A. Rabbou, R. Nouri, S. M. R. Beheshti, and S. Sakr. DREAM: Distributed RDF engine with adaptive query planner and minimal communication. *PVLDB*, 8(6):654–665, 2015.

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

90. S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. In *International Workshop on Practical and Scalable Semantic Systems (PSSS)*, pages 1–20, 2003.

91. S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In *International Workshop on Scalable Semantic Web Systems (SSWS)*, pages 94–109, 2009.

92. S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation, 2013. http://www.w3.org/TR/sparql11-query/.

93. A. Harth and S. Decker. Optimized index structures for querying RDF from the Web. In *Latin American Web Congress (LA-WEB)*, pages 71–80. IEEE, 2005.

94. A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *International Semantic Web Conference (ISWC)*, pages 211–224. Springer, 2007.

95. M. Hassan and S. K. Bansal. Data Partitioning Scheme for Efficient Distributed RDF Querying Using Apache Spark. In *International Conference on Semantic Computing (ICSC)*, pages 24–31. IEEE, 2019.

96. L. He, B. Shao, Y. Li, H. Xia, Y. Xiao, E. Chen, and L. J. Chen. Stylus: A Strongly-Typed Store for Serving Massive RDF Data. *PVLDB*, 11(2):203–216, 2017.

97. P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph. OWL 2 Web Ontology Language Primer. W3C Recommendation, 2012. https://www.w3.org/TR/owl2-primer/.

98. A. Hogan, J. L. Reutter, and A. Soto. In-Database Graph Analytics with Recursive SPARQL. In *International Semantic Web Conference (ISWC)*, pages 511–528. Springer, 2020.

99. A. Hogan, C. Riveros, C. Rojas, and A. Soto. A worst-case optimal join algorithm for SPARQL. In *International Semantic Web Conference (ISWC)*, pages 258–275. Springer, 2019.

100. A. Hogan, C. Riveros, C. Rojas, and A. Soto. Wikidata Graph Pattern Benchmark (WGPB) for RDF/SPARQL, 2019.

101. A. Hogenboom, F. Frasincar, and U. Kaymak. Ant colony optimization for RDF chain queries for decision support. *Expert Syst. Appl.*, 40(5):1555–1563, 2013.

102. A. Hogenboom, V. Milea, F. Frasincar, and U. Kaymak. RCQ-GA: RDF Chain Query Optimization Using Genetic Algorithms. In *E-Commerce and Web Technologies (EC-Web)*, pages 181–192. Springer, 2009.

103. K. Hose and R. Schenkel. WARP: Workload-aware replication and partitioning for RDF. In *ICDE Workshops*, pages 1–6, 2013.

104. J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.

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

106. Y. E. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. In *International Conference on Management of Data (SIGMOD)*, pages 9–22. ACM, 1987.

107. H. Jabeen, E. Haziiev, G. Sejdiu, and J. Lehmann. DISE: A Distributed in-Memory SPARQL Processing Engine over Tensor Data. In *International Conference on Semantic Computing (ICSC)*, pages 400–407. ACM, 2020.

108. L. Jachiet, P. Genevès, N. Gesbert, and N. Layaïda. On the Optimization of Recursive Relational Queries: Application to Graph Queries. In *SIGMOD International Conference on Management of Data (SIGMOD)*, pages 681–697. ACM, 2020.

109. F. T. Jamour, I. Abdelaziz, Y. Chen, and P. Kalnis. Matrix Algebra Framework for Portable, Scalable and Efficient Query Engines for RDF Graphs. In *EuroSys Conference*, pages 27:1–27:15. ACM, 2019.

110. M. Janik and K. Kochut. BRAHMS: a workbench RDF store and high performance memory system for semantic association discovery. In *International Semantic Web Conference (ISWC)*, pages 431–445. Springer, 2005.

111. D. Janke and S. Staab. Storing and Querying Semantic Data in the Cloud. In *Reasoning Web Summer School*, pages 173–222. Springer, 2018.

112. D. Janke, S. Staab, and M. Thimm. Koral: A Glass Box Profiling System for Individual Components of Distributed RDF Stores. In *Workshop on Benchmarking Linked Data (BLINK)*. CEUR, 2017.

113. D. Janke, S. Staab, and M. Thimm. On Data Placement Strategies in Distributed RDF Stores. In *International Workshop on Semantic Big Data (SBD)*, pages 1–6. ACM, 2017.

114. E. G. Kalayci, T. E. Kalayci, and D. Birant. An ant colony optimisation approach for optimising SPARQL queries by reordering triple patterns. *Inf. Syst.*, 50:51–68, 2015.

115. O. Kalinsky, O. Mishali, A. Hogan, Y. Etsion, and B. Kimelfeld. Efficiently Charting RDF. *CoRR*, abs/1811.10955, 2018.

116. Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *VLDB Journal*, 24(1):67–91, 2015.

117. G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: a declarative query language for RDF. In *International World Wide Web Conference (WWW)*, pages 592–603. ACM, 2002.

118. G. Karvounarakis, A. Magkanaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. Querying the Semantic Web with RQL. *Comput. Net.*, 42(5):617–640, 2003.

119. G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

120. A. Katib, V. Slavov, and P. Rao. RIQ: Fast processing of SPARQL queries on RDF quadruples. *Journal of Web Sem.*, 37:90–111, 2016.

121. V. Khadilkar, M. Kantarcioglu, B. Thuraisingham, and P. Castagna. Jena-HBase: A Distributed, Scalable and Efficient RDF Triple Store. In *ISWC Posters and Demos (ISWC-PD)*, page 85–88. CEUR, 2012.

122. J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi. Taming Subgraph Isomorphism for RDF Query Processing. *PVLDB*, 8(11), 2015.

123. K. Kim, B. Moon, and H. Kim. Rg-index: An RDF graph index for efficient SPARQL query processing. *Expert Syst. Appl.*, 41(10):4596–4607, 2014.

124. K. Kim, B. Moon, and H. Kim. R3F: RDF triple filtering method for efficient SPARQL query processing. *WWW J.*, 18(2):317–357, 2015.

125. A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM - A pragmatic semantic repository for OWL. In *WISE Workshops*, pages 182–192. Springer, 2005.

126. D. Kolas, I. Emmons, and M. Dean. Efficient linked-list RDF indexing in Parliament. In *International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, pages 17–32. CEUR, 2009.

127. A. Koschmieder and U. Leser. Regular Path Queries on Large Graphs. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, volume 7338 of *LNCS*, pages 177–194. Springer, 2012.

128. E. V. Kostylev, J. L. Reutter, M. Romero, and D. Vrgoc. SPARQL with Property Paths. In *International Semantic Web Conference (ISWC)*, pages 3–18. Springer, 2015.

129. M. Koubarakis and K. Kyzirakos. Modeling and Querying Metadata in the Semantic Sensor Web: The Model StRDF and the Query Language StSPARQL. In *Extended Semantic Web Conference (ESWC)*, pages 425–439. Springer, 2010.

130. K. Kyzirakos, M. Karpathiotakis, and M. Koubarakis. Strabon: A Semantic Geospatial DBMS. In *International Semantic Web Conference (ISWC)*, pages 295–311. Springer, 2012.

131. G. Ladwig and A. Harth. CumulusRDF: Linked Data Management on Nested Key-Value Stores, 2011.

132. T. Lampo, M. Vidal, J. Danilow, and E. Ruckhaus. To Cache or Not To Cache: The Effects of Warming Cache in Complex SPARQL Queries. In *On the Move to Meaningful Internet Systems (OTM)*, pages 716–733. Springer, 2011.

133. W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable Multi-Query Optimization for SPARQL. In *International Conference on Data Engineering (ICDE)*, ICDE '12, page 666–677, USA, 2012. IEEE.

134. K. Lee and L. Liu. Scaling queries over big RDF graphs with semantic hash partitioning. *PVLDB*, 6(14):1894–1905, 2013.

135. J. Leeka and S. Bedathur. RQ-RDF-3X: going beyond triplestores. In *ICDE Workshops*, pages 263–268. IEEE, 2014.

136. J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin, N. Chakraborty, M. Saleem, A.-C. Ngonga Ngomo, and H. Jabeen. Distributed Semantic Analytics Using the SANSA Stack. In *International Semantic Web Conference (ISWC)*, pages 147–155. Springer, 2017.

137. A. Letelier, J. Pérez, R. Pichler, and S. Skritek. Static analysis and optimization of semantic web queries. *ACM TODS*, 38(4):25:1–25:45, 2013.

138. W. Li, B. Zhang, G. Rao, R. Chen, and Z. Feng. Hash Tree Indexing for Fast SPARQL Query in Large Scale RDF Data Management Systems. In *ISWC Posters & Demonstrations and Industry Tracks*. CEUR-WS.org, 2017.

139. B. Liu and B. Hu. HPRD: a high performance RDF database. *Int. J. Parallel Emergent Distr. Syst.*, 25(2):123–133, 2010.

140. J. Lorey and F. Naumann. Caching and Prefetching Strategies for SPARQL Queries. In *ESWC Satellite Events*, pages 46–65. Springer, 2013.

141. Y. Luo, F. Picalausa, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. Storing and Indexing Massive RDF Datasets. In *Semantic Search over the Web*, pages 31–60. Springer, 2012.

142. X. Lyu, X. Wang, Y. Li, Z. Feng, and J. Wang. GraSS: An Efficient Method for RDF Subgraph Matching. In *Web Information Systems Engineering Conference (WISE)*, pages 108–122. Springer, 2015.

143. L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar: an RDF storage and query system for enterprise resource management. In *International Conference on Information and Knowledge Management (CIKM)*, pages 484–491, 2004.

144. Z. Ma, M. A. Capretz, and L. Yan. Storing massive Resource Description Framework (RDF) data: A survey. *Knowl. Eng. Rev,*, 31(4):391–413, 2016.

145. A. Maccioni and M. Collina. Graph databases in the browser: using LevelGraph to explore New Delhi. *Proc. VLDB Endow.*, 9(13):1469–1472, 2016.

146. A. Madkour, A. M. Aly, and W. G. Aref. WORQ: Workload-Driven RDF Query Processing. In *International Semantic Web Conference (ISWC)*, pages 583–599. Springer, 2018.

147. R. Maharjan, Y. Lee, and S. Lee. Exploiting Path Indexes to Answer Complex Queries in Ontology Repository. In *International Conference on Computational Science and Its Applications (ICCSA)*, pages 56–61. IEEE, 2009.

148. S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *International Semantic Web Conference (ISWC)*, pages 376–394. Springer, 2018.

149. W. Martens and T. Trautner. Evaluation and Enumeration Problems for Regular Path Queries. In *International Conference on Database Theory (ICDT)*, pages 19:1–19:21, 2018.

150. M. Martin, J. Unbehauen, and S. Auer. Improving the Performance of Semantic Web Applications with SPARQL Query Caching. In *Extended Semantic Web Conference (ESWC)*, pages 304–318. Springer, 2010.

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

152. J. P. McGlothlin and L. R. Khan. RDFJoin: A Scalable Data Model for Persistence and Efficient Querying of RDF Datasets. Tech. Rep. UTDCS-08-09, Univ. of Texas at Dallas, 2009.

153. J. P. McGlothlin and L. R. Khan. RDFKB: Efficient support for RDF inference queries and knowledge management. In *International Database Engineering & Applications Symposium (IDEAS)*, pages 259–266. ACM, 2009.

154. M. Meimaris and G. Papastefanatos. Distance-based triple reordering for SPARQL query optimization. In *International Conference on Data Engineering (ICDE)*, pages 1559–1562. IEEE Computer Society, 2017.

155. M. Meimaris, G. Papastefanatos, N. Mamoulis, and I. Anagnostopoulos. Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization. In *International Conference on Data Engineering (ICDE)*, pages 497–508. IEEE, 2017.

156. S. Metzler and P. Miettinen. On Defining SPARQL with Boolean Tensor Algebra. *CoRR*, abs/1503.00301, 2015.

157. T. Minier, H. Skaf-Molli, and P. Molli. SaGe: Web Preemption for Public SPARQL Query Services. In *World Wide Web Conference (WWW)*, pages 1268–1278. ACM, 2019.

158. K. Miura, T. Amagasa, and H. Kitagawa. Accelerating Regular Path Queries using FPGA. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB)*, pages 47–54, 2019.

159. M. Morsey, J. Lehmann, S. Auer, and A. N. Ngomo. DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data. In *International Semantic Web Conference (ISWC)*, pages 454–469. Springer, 2011.

160. R. Mutharaju, S. Sakr, A. Sala, and P. Hitzler. D-SPARQ: Distributed, Scalable and Efficient RDF Query Engine. In *ISWC Posters & Demos (ISWC-PC)*, page 261–264. CEUR, 2013.

161. A. Muys. Building an Enterprise-Scale Database for RDF Data. Netymon technical paper, 2006.

162. H. Naacke and O. Curé. On distributed SPARQL query processing using triangles of RDF triples. *Open J. Semantic Web*, 7(1):17–32, 2020.

163. G. Navarro, J. L. Reutter, and J. Rojas-Ledesma. Optimal Joins Using Compact Data Structures. In *International Conference on Database Theory (ICDT)*, pages 21:1–21:21. S. Dagstuhl, 2020.

164. Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. RDFox: A Highly-Scalable RDF Store. In *International Semantic Web Conference (ISWC)*, pages 3–20. Springer, 2015.

165. T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *International Conference on Data Engineering (ICDE)*, pages 984–994. IEEE, 2011.

166. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, 2008.

167. T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *International Conference on Management of Data (SIGMOD)*, pages 627–640. ACM, 2009.

168. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDBJ*, 19(1):91–113, 2010.

169. H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case Optimal Join Algorithms. *J. ACM*, 65(3):16:1–16:40, 2018.

170. V. Nguyen and K. Kim. Efficient Regular Path Query Evaluation by Splitting with Unit-Subquery Cost Matrix. *IEICE Trans. Inf. Syst.*, 100-D(10):2648–2652, 2017.

171. B. Nowack. http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-135/paper2.pdf. In *Scripting for the Semantic Web (SFSW@ESWC)*, volume 135, pages 1–8. CEUR-WS.org, 2006.

172. R. Oldakowski, C. Bizer, and D. Westphal. RAP: RDF API for PHP. In *Scripting for the Semantic Web*, 2005.

173. A. Owens, A. Seaborne, N. Gibbins, and m. schraefel. Clustered TDB: A clustered triple store for Jena. EPrints Server, 2008.

174. M. T. Özsu. A survey of RDF data management systems. *Frontiers of Comp. Sci.*, 10(3):418–432, 2016.

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

176. M.-E. Papadaki, N. Spyratos, and Y. Tzitzikas. Towards Interactive Analytics over RDF Graphs. *Algorithms*, 14(2), 2021.

177. N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF+: High-performance distributed joins over large-scale RDF graphs. *Big Data*, pages 255–263, 2013.

178. N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2RDF: Adaptive Query Processing on RDF Data in the Cloud. In *World Wide Web Conference (WWW)*, page 397–400. ACM, 2012.

179. N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris. Graph-Aware, Workload-Adaptive SPARQL Query Caching. In *International Conference on Management of Data (SIGMOD)*, pages 1777–1792. ACM, 2015.

180. P. Peng, Q. Ge, L. Zou, M. T. Özsu, Z. Xu, and D. Zhao. Optimizing Multi-Query Evaluation in Federated RDF Systems. *IEEE TKDE*, 33(4):1692–1707, 2021.

181. P. Peng, L. Zou, L. Chen, and D. Zhao. Adaptive Distributed RDF Graph Fragmentation and Allocation based on Query Workload. *IEEE TKDE*, 31(4):670–685, 2019.

182. P. Peng, L. Zou, M. T. Özsu, L. Chen, and D. Zhao. Processing SPARQL Queries over Distributed RDF Graphs. *VLDBJ*, 25(2):243–268, 2016.

183. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM TODS*, 34(3), 2009.

184. M. Pham and P. A. Boncz. Exploiting Emergent Schemas to Make RDF Systems More Efficient. In *International Semantic Web Conference (ISWC)*, pages 463–479, 2016.

185. G. E. Pibiri, R. Perego, and R. Venturini. Compressed Indexes for Fast Search of Semantic Data. *IEEE TKDE*, 33(9):3187–3198, 2021.

186. F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. A Structural Approach to Indexing Triples. In *Extended Semantic Web Conference (ESWC)*, pages 406–421. Springer, 2012.

187. A. Potocki, A. Polukhin, G. Drobyazko, D. Hladky, V. P. Klintsov, and J. Unbehauen. OntoQuad: Native High-Speed RDF DBMS for Semantic Web. In *Knowledge Engineering and the Semantic Web (KESW)*, volume 394 of *Communications in Computer and Information Science*, pages 117–131. Springer, 2013.

188. A. Potter, B. Motik, Y. Nenov, and I. Horrocks. Dynamic Data Exchange in Distributed RDF Stores. *IEEE TKDE*, 30(12):2312–2325, 2018.

189. R. Punnoose, A. Crainiceanu, and D. Rapp. Rya: A Scalable RDF Triple Store for the Clouds. In *International Workshop on Cloud Intelligence (Cloud-I)*. ACM, 2012.

190. S. Purohit, N. Van, and G. Chin. Semantic Property Graph for Scalable Knowledge Graph Analytics. *CoRR*, abs/2009.07410, 2020.

191. S. Qiao and Z. M. Özsoyoglu. RBench: Application-Specific RDF Benchmarking. In *International Conference on Management of Data (SIGMOD)*, pages 1825–1838. ACM, 2015.

192. A. Rajith, S. Nishimura, and H. Yokota. JARS: join-aware distributed RDF storage. In *International Database Engineering & Applications Symposium (IDEAS)*, pages 264–271. ACM, 2016.

193. P. Ravindra, H. Kim, and K. Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *Extended Semantic Web Conference (ESWC)*, pages 46–61. Springer, 2011.

194. J. L. Reutter, A. Soto, and D. Vrgoc. Recursion in SPARQL. In *International Semantic Web Conference (ISWC)*, pages 19–35. Springer, 2015.

195. K. Rohloff and R. E. Schantz. High-Performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The SHARD Triple-Store. In *Programming Support Innovations for Emerging Distributed Applications (PSI EtA)*. ACM, 2010.

196. S. Sakr and G. Al-Naymat. Relational processing of RDF queries: a survey. *SIGMOD Record*, 38(4):23–28, 2010.

197. M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A. N. Ngomo. LSQ: The Linked SPARQL Queries Dataset. In *International Semantic Web Conference (ISWC)*, pages 261–269. Springer, 2015.

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

199. M. Saleem, Q. Mehmood, and A. N. Ngomo. FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework. In *International Semantic Web Conference (ISWC)*, pages 52–69. Springer, 2015.

200. M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood, and A.-C. Ngonga Ngomo. How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks. In *World Wide Web Conference (WWW)*, page 1623–1633. ACM, 2019.

201. A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen. S2X: Graph-Parallel Querying of RDF with GraphX. In *Workshop on Biomedical Data Management and Graph Online Querying (Big-O(Q))*, pages 155–168. Springer, 2016.

202. A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *International Workshop on Semantic Web Information Management (SWIM)*, 2011.

203. A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen. Sempala: Interactive SPARQL query processing on Hadoop. In *International Semantic Web Conference (ISWC)*, pages 164–179, 2014.

204. A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF querying with SPARQL on Spark. *PVLDB*, 9(10):804–815, 2016.

205. M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In *International Semantic Web Conference (ISWC)*, pages 585–600. Springer, 2011.

206. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. In *International Conference on Data Engineering ICDE*, pages 222–233. IEEE, 2009.

207. M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *International Conference on Database Theory (ICDT)*, pages 4–33, 2010.

208. G. Schreiber and Y. Raimond. RDF 1.1 Primer. W3C Working Group Note, 2014. http://www.w3.org/TR/rdf11-primer/.

209. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *International Conference on Management of Data (SIGMOD)*, pages 23–34. ACM, 1979.

210. S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *International Conference on Data Engineering (ICDE)*, pages 1009–1020. IEEE, 2013.

211. J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *Conference on Operating Systems Design and Implementation (OSDI)*, pages 317–332. USENIX, 2016.

212. M. Sintek and M. Kiesel. RDFBroker: A Signature-Based High-Performance RDF Store. In *European Semantic Web Conference (ESWC)*, pages 363–377. Springer, 2006.

213. A. Sotona and S. Negru. How to feed Apache HBase with Petabytes of RDF Data: An Extremely Scalable RDF Store Based on Eclipse RDF4J Framework and Apache HBase Database. In *ISWC Posters & Demonstrations Track*, volume 1690. CEUR-WS.org, 2016.

214. C. Stadler, G. Sejdiu, D. Graux, and J. Lehmann. Sparklify: A Scalable Software Component for Efficient Evaluation of SPARQL Queries over Distributed RDF Datasets. In *International Semantic Web Conference (ISWC)*, pages 293–308. Springer, 2019.

215. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *World Wide Web Conference (WWW)*, pages 595–604. ACM, 2008.

216. M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column-Oriented DBMS. In *VLDB*, pages 553–564. VLDB End., 2005.

217. H. Stuckenschmidt. Similarity-Based Query Caching. In *International Conference on Flexible Query Answering Systems (FQAS)*, pages 295–306. Springer, 2004.

218. H. Stuckenschmidt, R. Vdovjak, J. Broekstra, and G. Houben. Towards distributed processing of RDF path queries. *Int. J. Web Eng. Tech.*, 2(2/3):207–230, 2005.

219. P. Stutz, A. Bernstein, and W. Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web . In *International Semantic Web Conference (ISWC)*, pages 764–780. Springer, 2010.

220. P. Stutz, M. Verman, L. Fischer, and A. Bernstein. TripleRush: A Fast and Scalable Triple Store. In *Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, page 50–65. CEUR, 2013.

221. M. Svoboda and I. Mlỳnková. Linked Data indexing methods: A survey. In *OTM Confederated International Conferences (OTM)*, pages 474–483. Springer, 2011.

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

223. H. Thakkar, R. Angles, M. Rodriguez, S. Mallette, and J. Lehmann. Let's build Bridges, not Walls: SPARQL Querying of TinkerPop Graph Databases with Sparql-Gremlin. In *International Conference on Semantic Computing (ICSC)*, pages 408–415. IEEE, 2020.

224. B. B. Thompson, M. Personick, and M. Cutcher. The Bigdata® RDF Graph Database. In *Linked Data Management*, pages 193–237. CRC Press, 2014.

225. T. Tran, G. Ladwig, and S. Rudolph. Managing Structured and Semistructured RDF Data Using Structure Indexes. *IEEE TKDE*, 25(9):2076–2089, 2013.

226. C. Tschanz. Query-Driven Index Partitioning for TripleRush. Bachelor Thesis, 2014.

227. P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. A. Boncz. Heuristics-based query optimisation for SPARQL. In *International Conference on Extending Database Technology (EDBT)*, pages 324–335. ACM, 2012.

228. O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: A Graph Based RDF Index. In *Conference on Artificial Intelligence (AAAI)*, pages 1465–1470. AAAI, 2007.

229. M. van Otterdijk, G. Mendel-Gleason, and K. Feeney. Succinct Data Structures and Delta Encoding for Modern Databases. TerminusDB White Paper, 2020. https://github.com/terminusdb/terminusdb/blob/main/docs/whitepaper/terminusdb.pdf.

230. T. L. Veldhuizen. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *International Conference on Database Theory (ICDT)*, pages 96–106. OpenProceedings.org, 2014.

231. M. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently Joining Group Patterns in SPARQL Queries. In *Extended Semantic Web Conference (ESWC)*, pages 228–242. Springer, 2010.

232. A. Vlachou, C. Doulkeridis, A. Glenis, G. M. Santipantakis, and G. A. Vouros. Efficient Spatio-Temporal RDF Query Processing in Large Dynamic Knowledge Bases. In *Symposium on Applied Computing (SAC)*, page 439–447. ACM, 2019.

233. D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *CACM*, 57(10):78–85, 2014.

234. S. Wadhwa, A. Prasad, S. Ranu, A. Bagchi, and S. Bedathur. Efficiently Answering Regular Simple Path Queries on Large Labeled Networks. In *SIGMOD International Conference on Management of Data*, pages 1463–1480. ACM, 2019.

235. S. Wang, C. Lou, R. Chen, and H. Chen. Fast and Concurrent RDF Queries Using RDMA-Assisted GPU Graph Exploration. In *USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 651–664, USA, 2018. USENIX.

236. X. Wang, L. Chai, Q. Xu, Y. Yang, J. Li, J. Wang, and Y. Chai. Efficient Subgraph Matching on Large RDF Graphs Using MapReduce. *Data Sci. Eng.*, 4(1):24–43, 2019.

237. X. Wang, S. Wang, Y. Xin, Y. Yang, J. Li, and X. Wang. Distributed Pregel-based provenance-aware regular path query processing on RDF knowledge graphs. *World Wide Web*, 23(3):1465–1496, 2020.

238. C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

239. R. T. Whitman, B. G. Marsh, M. B. Park, and E. G. Hoel. Distributed Spatial and Spatio-Temporal Join on Apache Spark. *ACM TSAS*, 5(1), 2019.

240. J. Wielemaker, W. Beek, M. Hildebrand, and J. van Ossenbruggen. ClioPatria: A SWI-Prolog infrastructure for the Semantic Web. *Semantic Web*, 7(5):529–541, 2016.

241. K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *International Conference on Semantic Web and Databases (SWDB)*, pages 120–139. CEUR, 2003.

242. G. T. Williams and J. Weaver. Enabling Fine-Grained HTTP Caching of SPARQL Query Results. In *International Semantic Web Conference (ISWC)*, pages 762–777. Springer, 2011.

243. D. Wood, P. Gearon, and T. Adams. Kowari: A platform for semantic web storage and analysis. In *XTech Conference*, pages 1–7, 2005.

244. B. Wu, Y. Zhou, P. Yuan, H. Jin, and L. Liu. SemStore: A Semantic-Preserving Distributed RDF Triple Store. In *Conference on Information and Knowledge Management (CIKM)*, pages 509–518. ACM, 2014.

245. G. Wu, J. Li, J. Hu, and K. Wang. System Pi: A Native RDF Repository Based on the Hypergraph Representation for RDF Data Model. *J. Comput. Sci. Technol.*, 24(4):652–664, 2009.

246. G. Wu and M. Yang. Improving SPARQL query performance with algebraic expression tree based caching and entity caching. *J. Zhejiang Univ. Sci. C*, 13(4):281–294, 2012.

247. H. Wu, T. Fujiwara, Y. Yamamoto, J. Bolleman, and A. Yamaguchi. BioBenchmark Toyama 2012: An evaluation of the performance of triple stores on biological data. *J. Bio. Sem.*, 5, 2014.

248. M. Wylot and P. Cudré-Mauroux. DiploCloud: Efficient and Scalable Management of RDF Data in the Cloud. *IEEE TKDE*, 28(3):659–674, 2016.

249. M. Wylot, M. Hauswirth, P. Cudré-Mauroux, and S. Sakr. RDF Data Storage and Query Processing Schemes: A Survey. *ACM CSUR*, 51(4):84:1–84:36, 2018.

250. M. Wylot, J. Pont, M. Wisniewski, and P. Cudré-Mauroux. DipLODocus[RDF]: Short and Long-Tail RDF Analytics for Massive Webs of Data. In *International Semantic Web Conference (ISWC)*, pages 778–793. Springer, 2011.

251. G. Xiao, D. Lanti, R. Kontchakov, S. Komla-Ebri, E. G. Kalayci, L. Ding, J. Corman, B. Cogrel, D. Calvanese, and E. Botoeva. The Virtual Knowledge Graph System Ontop. In *International Semantic Web Conference (ISWC)*, volume 12507, pages 259–277. Springer, 2020.

252. N. Yakovets, P. Godfrey, and J. Gryz. Evaluation of SPARQL Property Paths via Recursive SQL. In *Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*. CEUR, 2013.

253. N. Yakovets, P. Godfrey, and J. Gryz. Query Planning for Evaluating SPARQL Property Paths. In *International Conference on Management of Data (SIGMOD)*, pages 1875–1889. ACM, 2016.

254. S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *International Conference on Management of Data (SIGMOD)*, pages 517–528. ACM, 2012.

255. M. Q. Yasin, X. Zhang, R. Haq, Z. Feng, and S. Yitagesu. A Comprehensive Study for Essentiality of Graph Based Distributed SPARQL Query Processing. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 156–170. Springer, 2018.

256. P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: A Fast and Compact System for Large Scale RDF Data. *PVLDB*, 6(7):517–528, 2013.

257. L. H. Zambom Santana and R. dos Santos Mello. An Analysis of Mapping Strategies for Storing RDF Data into NoSQL Databases. In *Symposium on Applied Computing (SAC)*, pages 386–392. ACM, 2020.

258. K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. In *PVLDB*, pages 265–276, 2013.

259. L. Zervakis, V. Setty, C. Tryfonopoulos, and K. Hose. Efficient Continuous Multi-Query Processing over Graph Streams. In *International Conference on Extending Database Technology (EDBT)*, pages 13–24. OpenProceedings.org, 2020.

260. W. E. Zhang, Q. Z. Sheng, K. Taylor, and Y. Qin. Identifying and Caching Hot Triples for Efficient RDF Query Processing. In *Database Systems for Advanced Applications (DASFAA)*, pages 259–274. Springer, 2015.

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

262. W. Zheng, L. Zou, X. Lian, H. Zhang, W. Wang, and D. Zhao. SQBC: an efficient subgraph matching method over large and dense graphs. *Inf. Sci.*, 261:116–131, 2014.

263. L. Zou, J. Mo, L. Chen, M. Tamer Özsu, and D. Zhao. gStore: Answering SPARQL queries via subgraph matching. *PVLDB*, 4(8):482–493, 2011.

A Survey of RDF Stores

We now present a survey of local and distributed RDF stores, and how they use the aforementioned techniques. At the end of this section, we will discuss some general trends for RDF stores. We include here systems for which we could find technical details regarding (at least) the storage, indexing and processing of joins over RDF graphs.[7] In the case of distributed RDF stores, we expect similar technical details, along with the type of partitioning and/or replication used. We include systems with associated publications, as well as systems that are unpublished but widely known in practice. Both local and distributed systems are presented in approximate chronological order, based on the year of publication, or an approximate year in which the system was released. For unpublished local stores, we include the year where RDF was first supported. For unpublished distributed stores, we include the approximate year when distributed features were added. Some stores that are often deployed in local environments also support distribution; they are included in both sections. Some systems are unnamed; if it is a distributed store that extends an existing local store, we append the suffix "-D" or "-D2" to the local store's name; otherwise we use an abbreviation based on authors and year. Where systems change name, we prefer the more modern name. The papers sometimes use different terminology to refer to similar concepts; we often map the original terminology to that used in the body of the survey in order to increase coherency and improve readability.

A.1 Local RDF Stores

The local RDF stores we include, and the techniques they use, are summarized in Table 4.

Redland [28] (2001) is a set of RDF libraries for native RDF storage that has seen various developments over the years. The original paper describes triple-table like storage based on creating three hash maps – SP→O, PO→S, SO→P – which, given two elements of an RDF triple, allow for finding the third element; for example, using PO→S, we can find the subjects of triples with a given predicate and object. The hash maps can be stored either in-memory or on persistent storage. Support for the RDQL and SPARQL query languages were later added with the Rasqal query library.

---

[7] We thus exclude systems – such as TPF-based systems, and stores such as Fabric, Fluree, and TriplePlace [18] – that do not (yet) describe direct support for joins or basic graph patterns. We also exclude systems – such as Attean, Kineo, KiWi, librdf.sqlite, NitroBase, Oxigraph, Pointrel, Profium Sense, RedStore, RDF::Trine, TerminusDB [229] and TriplyDB – for which we could not find key technical details (e.g., indexes or join algorithms supported) at the time of writing.

Jena [151] (2002) uses relational databases to store RDF graphs as triple tables, with entries for subject, predicate, object IRIs, and object literals. IRIs and literals are encoded as IDs and two separate dictionaries are created for both. Indexing is delegated to an underlying relational DBMS (e.g., Postgresql, MySQL, Oracle, etc.). RDQL is used as a query language and is translated into SQL and run against the underlying relational DBMS. The Jena store would later be extended in various directions, with SDB referring to the use of relational-style storage (per the original system), and TDB referring to the use of native storage.

RDF4J [42] (2002), known originally as Sesame, provides persistent storage and querying of RDF data. RDF4J provides storage-independent solutions and can be deployed on top of a variety of storage engines such as RDBMSs and object-oriented databases. Graph queries can be expressed with the RQL language. The storage, indexing, and query processing techniques depend on the underlying storage engine used by RDF4J. Recent versions of RDF4J features improved functionalities such as both in-memory and persistent data storage, SeRQL and SPARQL support, etc.

RSSDB [117, 118] (2002) stores an RDF graph using a vertical partitioning approach with Postgres as the underlying database. Two variants are considered for class instances: creating a unary table per class (named after the class, with rows indicating instances), or creating one binary table called instances (with rows containing both the instance and the class) in order to reduce the number of tables. Four tables are also added to model RDFS definitions (classes, properties with their domain and range, sub-classes and sub-properties). The system supports queries in RQL (proposed in the same paper [118]), which are translated to SQL by an RQL interpreter and evaluated over Postgres.

3store [90] (2003) uses MySQL as a back-end, sorting RDF graphs in four tables, namely a triple table, a models table, a resource table, and a literal table. The triple table stores RDF triples (one per row) with additional information: (1) the model the triple belongs to, (2) a boolean value to indicate if the object is a literal, and (3) a boolean value to indicate if this triple is inferred. The models, resource, and literal tables are two-column tables that dictionary encode models, resources, and literals, respectively. Queries expressed in RDQL are rewritten to SQL for execution over MySQL.

AllegroGraph [8] (2003) is a general purpose store for semi-structured data that can be used to query documents (e.g., JSON) and graph data (e.g., RDF). RDF data is stored and indexed in six permutations as quads, which are additionally associated with a triple identifier. SPARQL queries are

---

[8] https://franz.com/agraph/allegrograph/

Table 4: Categorization of local RDF Engines.

Storage: T = Triple Table, Q = Quad Table, V = Vertical Partitioning, P = Property table, G = Graph-based, E = Matrix/Tensor-based, M = Miscellaneous
Indexing: T = Triple, Q = Quad, E = Entity, P = Property, N = Path/Navigational, J = Join, S = Structural, M = Miscellaneous
Join P.: P = Pairwise, M = Multiway, W = Worst case optimal, L = Linear algebra
Query P.: R = Relational, N = Paths/Navigational, Q = Query rewriting

Engine	Year	Storage T	Q	V	P	G	E	M	Indexing T	Q	E	P	N	J	S	M	Join P. P	M	W	L	Query P. R	N	Q
Redland [28]	2001	✓							✓														
Jena [151]	2002	✓														✓	✓						✓
RDF4J [42]	2002	✓		✓												✓	✓						✓
RSSDB [117,118]	2002	✓														✓	✓						✓
3store [90]	2003	✓														✓	✓						✓
AllegroGraph	2003		✓							✓							✓				✓	✓	
Jena2 [241]	2003	✓			✓											✓	✓	✓					✓
CORESE [57,58]	2004	✓							✓								✓				✓		
Jena TDB	2004	✓	✓						✓	✓							✓				✓	✓	
RStar [143]	2004	✓														✓	✓						✓
BRAHMS [110]	2005	✓							✓								✓					✓	
GraphDB [125,33]	2005	✓	✓						✓	✓							✓				✓	✓	
Mulgara [243]	2005		✓							✓							✓				✓	✓	
RAP [172]	2005	✓							✓							✓	✓				✓		✓
RDF_MATCH [55]	2005		✓		✓				✓	✓				✓		✓	✓	✓			✓		
YARS [93]	2005		✓						✓								✓				✓		
ARC [171]	2006	✓	✓						✓	✓							✓				✓		✓
RDFBroker [212]	2006				✓											✓	✓	✓			✓		
Virtuoso [69]	2006	✓	✓													✓	✓				✓	✓	
GRIN [228]	2007					✓									✓		✓						
SW-Store [1]	2007			✓																			
Blazegraph [224]	2008	✓	✓						✓	✓							✓	✓			✓	✓	
Hexastore [238]	2008				✓				✓								✓				✓		
RDF-3X [168]	2008	✓							✓								✓	✓			✓		
BitMat [21]	2009						✓		✓		✓									✓			
DOGMA [41]	2009				✓						✓				✓		✓						
LuposDate [77]	2009	✓							✓								✓						
Parliament [126]	2009	✓	✓						✓	✓							✓						
RDFJoin	2009	✓							✓					✓			✓	✓			✓		✓
System Π [245]	2009				✓				✓				✓				✓				✓		
HPRD [139]	2010	✓	✓						✓	✓		✓					✓	✓					
Stardog	2010		✓						✓								✓				✓	✓	
StrixDB	2010	✓							✓								✓						✓
dipLODocus [250]	2011				✓				✓		✓						✓				✓		
gStore [263]	2011				✓				✓		✓							✓					
SpiderStore [164]	2011				✓						✓												
SAINT-DB [186]	2012				✓										✓		✓						
Strabon [130]	2012			✓				✓	✓								✓	✓			✓		
BrightstarDB	2013	✓	✓						✓	✓							✓				✓	✓	
DB2RDF [36]	2013				✓						✓						✓	✓			✓		✓
OntoQuad [187]	2013		✓							✓							✓	✓			✓		
OSQP [225]	2013				✓										✓		✓				✓		
Triplebit [256]	2013						✓		✓								✓	✓					
R3F [123,124]	2014	✓			✓				✓			✓	✓				✓						
RQ-RDF-3X [135]	2014			✓						✓							✓				✓		
SQBC [262]	2014				✓						✓							✓					
WaterFowl [61]	2014	✓							✓								✓				✓		
GraSS [142]	2015				✓				✓		✓							✓					
k^2-triples [13]	2015			✓					✓									✓					
RDFCSA [40,39]	2015	✓							✓									✓					
RDFox [164]	2015	✓							✓									✓			✓	✓	
Turbo$_{HOM++}$ [122]	2015				✓						✓							✓					
ClioPatria [240]	2016		✓						✓								✓				✓	✓	✓
LevelGraph [145]	2016	✓							✓								✓	✓					
RIQ [120]	2016		✓		✓				✓							✓	✓	✓					
axonDB [154]	2017	✓							✓			✓					✓	✓					
HTStore [138]	2017				✓						✓												
Ontop [46,251]	2017							✓				✓					✓				✓		✓
Quadstore	2017		✓							✓						✓	✓				✓	✓	
AMBER [105]	2018				✓				✓									✓					
TripleID-Q [49]	2018	✓				✓			✓											✓	✓	✓	
Jena-LTJ [99]	2019	✓							✓										✓				
MAGiQ [109]	2019					✓			✓											✓			
BMatrix [38]	2020					✓			✓									✓					
Tentris [31]	2020					✓			✓									✓			✓	✓	
Ring [19]	2021	✓							✓									✓					

supported, where the most recent version provides an option for two query engines: SBQE, which is optimized for SPARQL 1.0-style queries, and MJQE, which features merge joins and caching techniques optimized for property paths.

Jena2 [241] (2003) is a revised version of the original Jena database schema, with support for both triple and property tables. Unlike the original version, IRIs and literals are stored directly in the tables, unless they exceed a certain length, in which case they are dictionary encoded by two separate tables; this allows filter operations to be directly performed on the triple and property tables, thus reducing dictionary lookups, but increasing storage sizes as string values are stored multiple times. Indexing is handled by an underlying relational database, and graph queries in RDQL are rewritten to SQL queries evaluated over the database.

CORESE [57,58] (2004) began as a search engine with path-finding functionality and inference [57], but was extended to support SPARQL query features [58]. CORESE models RDF graphs as conceptual graphs; for simplicity we discuss their methods in terms of the RDF model. RDF graphs are indexed according to the terms, enabling the efficient evaluation of triple patterns. Given a basic graph pattern, the triple patterns are reordered based on heuristics – such as the number of constants or filters associated with the triple pattern, or the number of variables bound by previous triple patterns in the order – as well as cardinality estimates. A nested-loop style algorithm is then applied to perform joins. Filters are evaluated as soon as possible to reduce intermediate results.

Jena TDB [9] (2004) is a native RDF store that has seen continuous development in the past decades. A TDB instance consists of three tables: a node table (a dictionary, allowing to encode/decode RDF terms to/from 8-byte identifiers), a triple/quad table (with dictionary-encoded terms), and a prefixes table (used to store common prefixes used for abbreviations). Storage is based on custom B+trees used to build indexes for various triple/quad permutations. Join processing uses pairwise (nested-loop) joins, with a variety of statistic- and heuristic-based methods available for join reordering. SPARQL 1.1 query processing is implemented in the custom Jena ARQ query processor. Jena TDB has become the recommended RDF store for Jena, with older relational-based storage (later named Jena SDB) having been deprecated.

RStar [143] (2004) stores (RDFS-style) ontology information and instance data using multiple relations in the IBM DB2 RDBMS. Five two-column tables are used to store ontological data (property dictionary, sub-property relations, class dictionary, sub-class relations, and domain and range relations). Another five two-column tables are used to store

instance-related data (literal dictionary, IRI dictionary, triples, class instances, namespace dictionary). RStar pushes indexing and other tasks to the underlying database. The RStar Query Language (RSQL) is used and translated into SQL.

BRAHMS [110] (2005) is an in-memory RDF store. The RDF graph is indexed in three hash tables – S→PO, O→SP, P→SO – which allow for finding triples that use a particular constant. The motivating use-case of BRAHMS is to find semantic associations – i.e., paths between two subject/object nodes – in large RDF graphs. This path-search functionality was implemented in BRAHMS using depth-first search and breadth-first search algorithms.

GraphDB [125,33] (2005) (formerly known as OWLIM) stores RDF graphs using a mix of triple and quad tables. In the most recent version, indexes are built for two triple permutations (POS and PSO) as well as a quad permutation (GPSO). Predicate lists (SP and OP) can also be indexed in order to quickly find the predicates associated with a given subject or object. Terms are dictionary encoded. Joins are reordered according to cardinality estimations. SPARQL 1.1 is supported, along with a wide range of other features, including spatial features, full-text indexing, inference, semantic similarity, integration with MongoDB, and more besides.

Mulgara [243,161] (2005), a fork of an earlier RDF store known as Kowari, implements native RDF storage in the form of quads tables using AVL trees. Dictionary encoding based on 64-bit longs is used. Support for transactions is provided using immutable arrays that store quads on disk in compressed form, with skiplists enabling fast search; insertions and deletions lead to a new immutable array being generated on disk. Indexing is based on six permutations of quads (which is sufficient to efficiently evaluate all sixteen possible quad patterns). Joins are evaluated pairwise and reordered (possibly on-the-fly) based on cardinality estimations. Queries are expressed in the iTQL language, where SPARQL support was added later.

RAP [172] (2005) is a general-purpose PHP-based API for RDF that includes an RDF store. Two forms of storage are provided. An in-memory store collects triples in an array, with three indexes provided on S, P and O to find triples by a given term and position. Alternatively, persistent storage is supported through database backends, where triples are stored in a triple table in raw form (i.e., as RDF terms). Early versions of RAP supported RDQL queries. To evaluate such queries over the in-memory store, individual triple patterns are first evaluated and their results joined. If persistent storage is rather used, the query is rewritten in full to SQL and evaluated by the underlying database. Later versions would include support for SPARQL.

---

[9] https://jena.apache.org/documentation/tdb/

RDF_MATCH [55] (2005) is an RDF store that uses the Oracle RDBMS as an underlying database. It stores RDF data in two different tables: a dictionary table, and a quads table. Indexes are defined based on B-trees. Queries are evaluated as self-joins over the quads table, which is further joined with the dictionary. Materialized views can further be used to index s–s, s–p, s–o, p–p, p–o and o–o joins on-demand, as well as selected property tables. Support for Datalog-style rules is also provided. Queries are expressed in an SQL-style syntax, with functions used to express graph patterns, which in turn are interpreted and evaluated by Oracle.

YARS [93] (2005) is a native RDF store that indexes quad tables in B+trees using dictionary encoded terms. It uses four types of indexes: dictionary indexes, keyword indexes, quad indexes (with six permutations), and cardinality indexes that count occurrences of quad patterns. YARS implements pairwise (nested-loop) joins that are reordered by cardinality. Basic graph patterns in Notation3 (N3) syntax are supported.

ARC [171] (2006) was first proposed as a general-purpose PHP library for processing RDF, but would be extended in subsequent years with a variety of additional features, including RDF storage and querying. These features are implemented on top of an underlying relational store, with support for persistent storage through MariaDB and MySQL, and in-memory storage with SQLite. Data are stored as triple or quad tables. A fragment of the SPARQL standard is supported, including some selected features from SPARQL 1.1.

RDFBroker [212] (2006) is an RDF store that follows a property table approach. For each subject in the graph, its signature (equivalent to the notion of characteristic sets that would come later) is extracted, with a property table defined for each signature, including a column for the subject, and a column for each property in the signature. Support for RDFS reasoning is also described. An index over signatures is proposed based on a lattice that models set containment between signatures. Given a signature extracted from the query, the lattice can be used to find tables corresponding to signatures that subsume that of the query. A prototype based on in-memory storage is described, implementing typical relational query optimizations such as join reordering.

Virtuoso [69] (2006) stores RDF data as a quad table, where in the most recent version, by default, the quad table includes five indexes: PSOG (for P, PS, PSO and PSOG prefixes), POGS (for PO and POG), SP (for S), OP (for O), and GS (for G). The table is sorted by the primary key PSOG. If only the subject or object are known, the SP/OP index can be used to identify predicates, allowing the PSOG/POGS index to be subsequently used. In the case that only the graph is known, the GS index is joined with SP and then with PSOG. Subjects, predicates and graph names are dictionary encoded; objects

are stored in raw form (for fast filtering). Configurations for row-wise and column-wise storage are provided. Query execution is based on translating SPARQL queries into SQL to be executed on a custom underlying database.

GRIN [228] (2007) is an RDF store based on a structural index (see Section 5.6). This index is a binary tree, where the root refers to all the nodes of the graph, and both children divide the nodes of its parents based on a given distance from a given node in the graph. The leaves can then be seen as forming a partition of the triples in the graph induced by the nodes in its division. The structural index is used to find small subgraphs that may generate results for a query, over which an existing subgraph matching algorithm is applied.

SW-Store [2,1] (2007) is an RDF store based on vertical partitioning. SW-Store relies on a column-oriented DBMS called C-store [216], which is shown to pair well with vertical partitioning in terms of performance (e.g., the object column of a `foaf:age` table will have integers in an interval $[0, 150]$, which are highly compressible). Each table is indexed by both subject and object. An "overflow" triple table is used for inserts alongside the compressed, vertically partitioned tables. Jena ARQ is used to translate SPARQL queries into SQL for evaluation over C-Store. Pairwise joins are used, preferring merge joins when data are sorted appropriately, otherwise using index nested-loop joins. Materialization of s–o joins is also discussed.

Blazegraph [224] (2008), formerly known as *BigData*, is a native RDF store supporting SPARQL 1.1. Blazegraph allows for either indexing triples to store RDF graphs, or quads to store SPARQL datasets. Three index permutations are generated for triples, and six permutations are generated for quads; indexes are based on B+trees. Both row and column data storage models are supported, which can be saved both in-memory or on-disk. Dictionary encoding with 64-bit integers is used for compressed representation of RDF triples. Two query optimization strategies are available: the default approach uses static analysis and cardinality estimation; the second approach uses runtime sampling of join graphs. Supported joins include hash joins, index nested-loop joins, merge joins, and multiway star joins.

Hexastore [238] (2008) is an in-memory RDF store based on adjacency lists similar to Figure 8. Six indexes are built for all $3! = 6$ permutations of the elements of a triple. For example, in the SPO index, each subject s is associated with an ordered vector of predicates, wherein each p in turn points to an ordered vector of objects. In the PSO index, each p points to a vector of subjects, wherein each s points to the same vector of objects as used for sp in the SPO index. Terms are dictionary encoded. Having all six index orders allows for (pairwise) merge joins to be used extensively.

RDF-3X [166,167,168] (2008) stores RDF graphs as triple tables in compressed indexes, which are based on clustered B+trees whose values are delta-encoded to reduce space. Triples are indexed in all six possible ways. RDF-3X also indexes counts for all triple patterns with one or two (distinct) constants; thus, it can find the exact number of triples matching, for example, (**s**,*p*,*o*), (*s*,**p**,**o**), etc. Counts are also maintained for frequent path and star joins. Join reordering is based on a dynamic programming procedure, which uses the aforementioned counts as cardinality statistics, and leverages the complete indexing of all six permutations to enable merge joins, further evaluating multiway joins in the case of star patterns. A later extension adds sideways information passing strategies [167] in order to filter intermediate results based on global heuristics.

BitMat [21,20] (2009) considers a one-hot encoding of an RDF graph, i.e., a 3-dimensional bit array (or matrix/tensor) of dimension $|S| \times |P| \times |O|$, where S, P, O, indicate the set of subjects, predicates and objects of the graph; and where index s, p, o contains a 1 if the corresponding (dictionary encoded) triple (s, p, o) exists, or 0 otherwise. The system stores slices of this array (called BitMats), where for each predicate, SO and OS BitMats are stored; for each subject, a PO BitMat is stored; and for each object, a PS BitMat is stored. Each BitMat is a 2-dimensional bit array; e.g., the OS matrix of a given predicate p enables finding all objects matching **o** in $(\mathbf{s}, p, \mathbf{o})$ or all subjects matching **s** in $(\mathbf{s}, p, o)$. Though OP and SP BitMats could be indexed for subjects and objects, resp., the authors argue they would be rarely used. BitMats also store the count of 1's (triples) they contain, a row vector indicating which columns contain a 1, and a column vector indicating which rows contain a 1. In total, $2|P||S||O|$ BitMats are generated, gap compressed, stored on disk, and loaded in memory as needed. Bitwise AND/OR/NOT operators are used for multiway joins.

DOGMA [41] (2009) is a graph-based RDF store, where an RDF graph is first decomposed into subgraphs using a graph partitioning algorithm. These subgraphs are indexed as the leaves of a balanced binary tree stored on disk. Each non-leaf node in this tree encodes the k-merge of its two children, which is a graph with k nodes that is isomorphic to a quotient graph (see Section 5.6) of both children. DOGMA proposes a variety of algorithms for evaluating basic graph patterns with constant predicates. The basic algorithm generates a set of candidate results for each individual variable node based on its incoming and outgoing edges; starting with the node with the fewest candidates, the algorithm then proceeds to check the edges between them in a depth-first manner (similar to wco joins). Further algorithms prune the sets of candidates based on their distance from the candidates of other nodes in the query based on the distance between nodes in the subgraphs in the leaves of the binary tree.

LuposDate [77] (2009) stores RDF in a triple table. Seven hash indexes are added for S, P, O, SP, SO, PO, SPO, enabling efficient evaluation of all eight triple patterns. Triples are also annotated with their rank (position in the order) with respect to the six permutations of the triple; for example, the OPS rank indicates the position of the triple when the graph is sorted in OPS order. These ranks are used for fast sorting of intermediate results when applying sort-merge joins.

Parliament [126] (2009) stores RDF graphs in three tables: a resource table encoding details of individual terms, a statement table encoding triples, and a dictionary table. These tables are stored as linked lists. For each RDF term, the resource table stores the first triple containing the term in the statement table; the number of triples that use it in the subject, predicate and object position; a pointer to its entry in the dictionary; and a bitvector encoding metadata about the term. The statement table contains eight components: a statement identifier; three identifiers for the subject, predicate, and object of the triple; three statement identifiers pointing to the next triple with the same subject, predicate, and object; and a bitvector for encoding metadata of the statement. This storage scheme avoids the need for multiple orders and enables fast lookups when triple patterns have one constant; for triple patterns with multiple constants, however, the most selective constant is looked up, with filters run to check the other constants. SPARQL query processing is enabled through existing libraries, such as Jena ARQ.

RDFJoin [152] (2009) stores RDF graphs using three types of tables. Two dictionary tables are used to encode and decode subjects/objects and predicates. Three triple tables are used, where each has two positions of the triple as the primary key, and the third position is encoded as a bit vector; for example, in the PO table, predicate and object are used as a primary key, and for each predicate–object pair, a bit vector of length $|so(G)|$ is given, with a 1 at index k encoding a triple with the subject identifier k for the given predicate and object. Join tables store the results of S–S, O–O, and S–O joins, encoded with the two predicates of both triples as primary key (joins using the same predicate twice are excluded), and a bit vector to encode the join terms for that predicate pair (the subject/object that matches the join variable). MonetDB and LucidDB are used as underlying databases. SPARQL is supported, where joins are evaluated using the join indexes and pairwise algorithms. Inference would later be added in the extended RDFKB system [153].

System Π [245] (2009) is a graph-based RDF store. Nodes of the graph are indexed with an identifier, value and type, while edges are indexed as triples, with their subject, predicate and object identifiers. Nodes are then linked to their inward and outward edges, which enables lookups for triples with a given subject or object. A more compressed version where outward edges only store the predicate and object, or inward edges only store the subject and predicate, is also proposed. Edges are then grouped by their vertex (outward edges by subject, inward edges by object). The compressed variant is thus similar to an adjacency list. Indexes are built for three triple permutations and for reachability queries on a single predicate (PLSD; see Section 5.4). SPARQL (1.0) queries are supported over the proposed indexes using pairwise joins, with the PLSD index used to support entailment over transitive properties.

HPRD [139] (2010) is an RDF store based on three types of index over the dictionary-encoded graph. Triple indexes are based on B+trees and cover three triple permutations: SPO, PO, OS. A path index is built using suffix arrays, and is used only to cache paths that are commonly accessed in queries; the paths are indexed by their predicates. Context indexes are based on B+Trees and are used to support temporal data, versioning, or named graphs; six permutations are covered, namely CSPO, SPO, POC, OCS, CP and SO. Cardinality statistics are further stored for triple patterns, and used for greedy join reordering. RDQL queries supported.

Stardog [10] (2010) is a commercial RDF store built upon the RocksDb key-value store. Stardog indexes quads in various permutations using RocksDB. Different types of pairwise joins – such as hash join, bind join, merge join etc. – are used. Stardog supports SPARQL 1.1, full-text search through Lucene, ACID transactions, versioning, and a variety of other features, including support for property graphs.

StrixDB [11] (2010) is a native RDF store with transactional support. Storage is built upon the kernel of Gigabase, which is an object-relational embedded database. One write transaction is supported in combination with multiple read transactions. Triples are indexed in B-trees under three permutations: SPO, POS and OSP. SPARQL queries are translated into executable byte-code. Rules in Datalog or Turtle-like syntax are supported though a variant of SLG resolution.

dipLODocus [250] (2011) is an RDF store based on the notion of a "molecule", which is a subgraph surrounding a particular "root" node. The root nodes are defined based on matching triple patterns provided by the administrator. The molecule of a root node is then the subgraph formed by

expanding outward in the graph until another root node is encountered. Dictionary encoding is used. Indexes are further built that map nodes and the values of properties indicated by the administrator to individual molecules. SPARQL is supported through the Rasqal query library, with joins pushed within individual molecules where possible; otherwise hash joins are used. Aggregate queries are pushed to the indexes on values of individual properties (which offers benefits similar to column-wise storage).

gStore [263] (2011) is a graph-based RDF store. The RDF graph is stored using adjacency lists (see Section 4.5) where each node is associated with a bit vector – which serves as a vertex signature (see Section 5.2) – that encodes the triples where the given node is the subject. gStore then indexes these signatures in a vertex signature tree (VS-tree) that enables multi-way joins. The leaves of the VS-tree encode signatures of nodes, and non-leaf nodes encode the bitwise OR of their children; the leaves are further connected with labeled edges corresponding to edges between their corresponding nodes in the graph. Basic graph patterns can then be encoded in a similar manner to the graph, where gStore then evaluates the pattern by matching its signature with that of the indexed graph.

SpiderStore [164] (2011) is an in-memory graph store based on adjacency lists. Specifically, for each node in the RDF graph, an adjacency list for incoming and outgoing edges is stored. Likewise, for each predicate, a list of subject nodes is stored. Rather than storing the constants directly in these lists, pointers are stored to the location of the term (with the adjacency lists for the node or the subjects of the predicate). Alongside these pointers, cardinality metadata are stored. (Though SPARQL queries with basic graph patterns and filters are evaluated in the experiments, the types of join algorithms used are not described.)

SAINT-DB [186] (2012) is an RDF store with a structural index that organizes triples in the graph according to the type of join that exists between them (S–S, P–O, etc.). The index itself is then a directed edge-labeled graph whose nodes represent a set of triples from the graph, edges indicate that some pair of triples in both nodes are joinable, and edge labels indicate the type of join that exists (which makes the graph directed as S–O differs from O–S). The nodes of the index then form a partition of the graph: no triple appears in more than one node, and their union yields the graph. This index can range from a single node with all triples in the graph (with loops for each type of join present), to singleton nodes each with one triple of the graph. A condition based on semi-joins is used to strike a balance, minimizing the intermediate results generated for individual triple patterns. Given a basic graph pattern, each triple pattern is then

mapped to nodes in the structural index, where the triple patterns it joins with must match some triple in a neighbor on an edge whose label corresponds to the type of join.

Strabon [130] (2012) is an RDF store that supports custom features for indexing and querying geospatial data (specifically in the form of stRDF [129] data). Strabon is built upon Sesame/RDF4J, which is chosen as an open-source solution that can easily integrate with PostGIS: a DBMS with spatial features. Strabon then stores RDF using a vertical partitioning scheme with dictionary encoding; an identifier for each triple is also included. B+tree indexes are built for the three columns of each table (subject, predicate, identifier). Strabon supports an extension of SPARQL, called stSPARQL [129], for querying stRDF based datasets, with spatial features supported through PostGIS.

BrightstarDB [12] (2013) is a persistent RDF store that indexes dictionary-encoded RDF datasets using B-trees and/or B+trees. Two types of persistence are supported: in append-only mode, writes are made to pages at the end of the index files, while in rewritable mode, writes are made to copies of index pages that are made active upon a commit. The system further supports querying over multiple named graphs. SPARQL 1.1 queries are processed over BrightstarDB's storage using dotNetRDF's Leviathan library, which supports hash joins and uses a heuristic-based join reordering based on which elements of the triple patterns are constant.

DB2RDF [36] (2013) uses a relational schema similar to property tables to store RDF data. However, rather than having a column for each property/predicate associated with a given subject, DB2RDF uses a "primary hash" table with columns S, P_1, O_1, ..., P_k, O_k, where each P_i, O_i pair of columns indicates the ith predicate–object pair associated with the subject listed in the S column. A binary "spill" column is added, with a 1 indicating that a subject has more than k triples, in which case it will occupy more than one row of the table. Rows for subjects with fewer than k triples are completed with NULLs. A second table is used to deal with multi-valued properties, where if a subject s has multiple values for the property p – say $o_1, \ldots, o_n$ – then a single fresh value v is used in the primary hash table, which is mapped to $o_1, \ldots, o_n$ (as a binary relation) in the second table. Two orders are indexed: in the second order, subjects and objects are reversed. IBM DB2 is used as the underlying database for experiments, with SPARQL queries being optimized and translated to SQL.

OntoQuad [187] (2013) is an RDF store that extends the triple-based representation of Hexastore to additionally support quads. A structure similar to a trie is used, where the top layer is a vector of values for S, P, O, G; the second level encodes SP, ..., GO, etc., with three children for each parent in the top layer (e.g., SP, SO, SG for S); the third layer has two children for each parent in the second layer encoding SPO, ..., GOP; the fourth layer has one child for each parent in the third layer, completing the quad permutation. B-trees are then used for indexing. Both pairwise and multiway joins are supported using zig-zag joins that seek forward to the maximum compatible join value across the triple patterns. Cardinality estimates and query rewriting rules are used to optimize SPARQL query plans.

OSQP [225] (2013) is an RDF store based on a structural index using various notions of bisimulation, where two nodes in the graph are bisimilar if they cannot be distinguished from their paths. The nodes of the graph are then partitioned into sets that are pairwise bisimilar. The index is then based on a quotient graph, where supernodes correspond to a set of bisimilar nodes in the input graph. In order to reduce index sizes, a parameter corresponding to path lengths is added, such that bisimulation only considers paths within a bounded region of the graph rather than the entire graph. A basic graph pattern is then matched over the quotient graph (kept in-memory), where the triples corresponding to each matched node are retrieved (from the disk) and used to compute the final results. Custom optimizations are considered for triples with unprojected variables, whose triple patterns can be definitively "satisfied" and thus pruned based on the index; and selective triple patterns, which are evaluated directly over the RDF graph.

Triplebit [256] (2013) represents a dictionary-encoded RDF graph as a compressed 2-dimensional bit matrix. Each column of the matrix represents a triple, and each row represents a subject/object node. The subject and object rows are assigned 1 for the corresponding column of the triple. Columns are sorted by predicate, where a range of columns corresponds to the triples for that predicate. The columns for triples are sparse (at most two 1's) and thus the two identifiers for subjects and objects are used, rather than storing 1's; two orders are maintained for SO and OS (thus effectively covering PSO and POS orders). Two auxiliary indexes are used in TripleBit. Given a subject or object node and a predicate node, the first index (called *ID-Chunk*) supports lookups for finding the range for the unspecified object or subject. Given a subject or object node alone, the second index (called *ID-predicate*) finds predicates associated with that subject or object. Basic graph patterns are evaluated using multiway merge-joins for star joins, with semi-joins used to reduce the number of intermediate results across star joins. Join ordering uses a greedy strategy on selectivity.

R3F [123, 124] (2014) is an extension of RDF-3X with path-based indexes and novel join processing techniques. The first addition is the "RP-index", which indexes all nodes with a given incoming path expression up to a certain length; for example, the incoming path expression pqr (of length 3) indexes all nodes z such that there exists w, x, y, such that $(w, p, x), (x, q, y), (y, r, z)$ are all triples of the graph. The RP-index is structured as a trie indexing the prefixes of the incoming path expressions, whose leaves are the list of nodes (which are dictionary encoded, sorted and delta encoded). Virtual inverse predicates are added to the RDF graph to support paths in both directions. The second extension is a modification to the sideways information passing strategy of RDF-3X to incorporate information about paths for filtering additional intermediate results.

RQ-RDF-3X [135] (2014) is an extension of RDF-3X towards support for quads. The extension follows the same principles and techniques for RDF-3X, but the extension to quads requires covering additional permutations. Indexes are built for all $4! = 24$ quad permutations, similar to how RDF-3X indexes all $3! = 6$ triple permutations; having all permutations enables reading the results for any variable of any triple pattern in sorted order, which in turn enables merge joins. The delta encoding used by RDF-3X is extended to the fourth element. Like in RDF-3X, counts are indexed for all quad patterns with 1, 2, or 3 constants, requiring 4, 12 and 24 indexes, respectively (40 in total). Join and query processing use RDF-3X's techniques. RQ-RDF-3X then offers optimized support for reification using named graphs/triple identifiers.

SQBC [262] (2014) is a graph store – with support for RDF graphs – inspired by existing subgraph matching techniques for efficiently finding subgraph isomorphisms.[13] In order to index the graph, codes are extracted for each node that capture structural information about it, including its label, the largest clique containing it, the degrees of its neighbours, etc. Given a basic graph pattern, candidates are identified and filtered for variable nodes. If the basic graph pattern has no cliques, degree information is used; otherwise clique sizes can be used to filter candidate matches.

WaterFowl [61] (2014) is a compact RDF store based on succinct data structures. The RDF graph is dictionary encoded and sorted in SPO order, and represented as a trie: the first layer denotes subjects, connected to their predicates in a second layer, connected to their objects in the third layer. This trie structure is encoded in a compact representation

---

[13] The evaluation of basic graph patterns in SPARQL is defined in terms of homomorphisms, rather than subgraph isomorphisms as supported by SQBC, with the difference being that two or more variables in a basic graph pattern can match one node in the RDF graph.

using a combination of bit strings that indicate the number of children for a parent (e.g., for predicates, 100101... tells us that the first subject has three children (unique predicates) and the second has two); and wavelet trees that encode the sequence of terms themselves (e.g., the sequence of predicates). Pairwise joins are evaluated in terms of left-deep plans, with further support for SPARQL (1.0) features. RDFS inference is also supported.

GraSS [142] (2015) is an RDF store that is based on decomposing basic graph patterns into subgraph patterns forming star joins (considering S–S, S–O, or O–O joins). An "FFD-index" for star joins is proposed, where for each node, a bitstring signature is computed that encodes its incoming and outgoing edges, i.e., the triples in which it appears as subject or object. A neighbourhood table is constructed: each row denotes a node, which is associated with its signature and edges. Five triple permutations are further indexed (covering SP*, OP*, S*, P*, O*), where in the SP* permutation, for example, (s, p) pairs are mapped to a list of objects and their degrees. A basic graph pattern is then decomposed into sub-patterns forming star joins, which are evaluated using the available indexes.

k^2-triples [13] (2015) is a compact in-memory RDF store based on k^2 trees. The RDF graph is first dictionary encoded. For each predicate, a k^2 tree is used to index its subjects and objects. In order to support variable predicates in triple patterns, SP and OP indexes are used to map subjects and objects, respectively, to their associated predicates; these indexes are encoded using compressed predicate lists. For processing basic graph patterns, S–S, S–O, O–S and O–O index nested-loop joins and merge joins are supported. A sideways information passing optimization is supported where two k^2 trees involved in a join can be descended in a coordinated fashion to filter intermediate results.

RDFCSA [40, 39] (2015) is a compact in-memory RDF store based on text indexes. Specifically, triples of the RDF graph are dictionary encoded and considered to be strings of length 3. The graph is thus sorted and encoded as a string of length $3n$, where n is the number of triples. This string is indexed in a compressed suffix array (CSA): a compact data structure commonly used for indexing text. The CSA is modified by shifting elements so that instead of indexing a string of $3n$ elements, triples cycle back on themselves, giving n circular strings of length 3. Thus in an SPO permutation, after reading the object of a triple, the next integer will refer to the subject of the same triple rather than the next one in the order. With cyclical strings, one triple permutation is sufficient to support all triple patterns; SPO is in fact equivalent to POS and OSP. Merge joins, sort–merge joins and a variant of index nested-loop joins (called "chain joins") are supported.

RDFox [164] (2015) is an in-memory RDF engine that supports Datalog reasoning. The RDF graph is stored as a triple table implemented as a linked list, which stores identifiers for subject, predicate and object, as well as three pointers in the list to the next triple with the same subject, predicate and object (similar to Parliament [126]). Four indexes are built: a hash table for three constants, and three for individual constants; the indexes for individual constants offer pointers to the first triple in the list with that constant, where patterns with two constants can be implemented by filtering over this list, or (optionally) by using various orderings of the triple list to avoid filtering (e.g., a triple list ordered by SPO can be used to evaluate patterns with constant subject and predicate without filtering). These in-memory indexes support efficient parallel updates, which are key for fast materialization. According to the implementation, (index) nested-loop joins are supported; optionally join plans can be generated based on tree decompositions. SPARQL 1.1 is further supported over the engine.

Turbo$_{HOM++}$ [122] (2015) is an in-memory, graph-based RDF store. The RDF graph is stored as the combination of adjacency lists for incoming and outgoing triples (see Section 4.5), and an index that allows for finding nodes of a particular type (based on `rdf:type`). Evaluation of basic graph patterns is then conducted by generating candidates for an initial node of the query graph based on local information (intersecting adjacency lists and type information in order to match all triple patterns that the node appears in), where the neighbors of the candidates are explored recursively in the graph guided by the graph pattern, generating candidates for further query nodes (in a manner akin to DOGMA [41]). A number of optimizations are included, including a multi-way join that can check if a new candidate is connected to the candidates of multiple query nodes in one operation in a way that satisfies the query.

ClioPatria [240] (2016) is an RDF store based on SWI-Prolog. RDF quads are stored in SWI-Prolog's main memory store. Nine custom hash-based indexes are defined using S, P, O, SP, PO, SPO, G, SG and PG as keys. Persistence is enabled though on-disk journal files. SPARQL queries are supported through rewriting to Prolog, and evaluated using SLD resolution. Inference can also be supported through translation of rules to Prolog.

LevelGraph [145] (2016) is an RDF store that can be used client-side (in-the-browser) with, for example, Node.js applications. LevelDB is used for underlying storage. Triples are stored and indexed in all six permutations – namely SPO, SOP, PSO, POS, OSP and OPS – which are stored in a sorted key–value store. LevelGraph also provides adapters to work with external NoSQL stores, such as DynamoDB, Redis,

MongoDB, and MySQL. A query optimizer supports merge joins (on star-shared patterns) and nested loop joins, which are used to evaluate basic graph graphs.

RIQ [120] (2016) provides a layer on top of an existing RDF store that indexes similar named graphs in a SPARQL dataset. A bit vector – called a "pattern vector" – is computed for each named graph in the dataset. The pattern vector consists of seven vectors for S, P, O, SP, SO, PO and SPO, where, e.g., the SP vector hashes all subject–predicate pairs in the named graph. An index over the pattern vectors (PV-index) is constructed by connecting similar pattern vectors (based on locality-sensitive hashing) into a graph; each connected component of the graph forms a group of similar graphs. The union of the graphs in each group is further encoded into Bloom filters. In order to evaluate a basic graph pattern, a pattern vector is computed combining the triple patterns (e.g., a triple pattern $(s, p, \mathbf{o})$ will generate a single SP sub-vector). The PV-index is then used to optimize an input query by narrowing down the candidate (named) graphs that match particular basic graph patterns before evaluating the optimized query over the underlying SPARQL store.

axonDB [154] (2017) uses two dictionary-encoded triple tables to store RDF graphs. In the first table, each triple is additionally associated with the characteristic set (CS) of its subject (see Section 5.3). The CS is assigned a unique identifier and one-hot encoded, i.e., represented by a bit vector with an index for each property that carries a 1 if the property is part of the CS, or a 0 otherwise. Triples are then sorted by their CS, grouping subjects with the same CS together. A second triple table stores each triple, along with the corresponding extended characteristic set (ECS; again see Section 5.3). The ECS is encoded with a unique identifier, and the identifiers for the subject and object CSs. The triple table is sorted by ECS. When evaluating a basic graph pattern, its analogous CSs and ECSs are extracted, along with the paths that connect them. The CSs and ECSs are matched with those of the graph, enabling multiway joins; binary hash joins are used to join the results of multiple CSs/ECSs.

HTStore [138] (2017) uses hash-based indexes to build an RDF store. The RDF graph is indexed in a hash tree whose top layer forms a hash table over the nodes of the graph. The hash tree is based on a sequence of prime numbers. When hashing a node, the first prime number is used, and if no collision is detected, the node is inserted in the first layer. Otherwise the second prime number is used, and if no collision is detected, it is inserted in that layer as a child of the bucket of the first layer that caused the collision. Otherwise the third prime number is used, and so forth. Nodes in the hash tree then point to their adjacency lists in the graph. To evaluate queries, constant nodes in the query are hashed in

the same manner in order to retrieve the data for the node. SPARQL queries are supported, though details about join and query processing are omitted.

Ontop [46] (2017) is a open source Ontology-Based Data Access (OBDA) system based on relational (and potentially decentralised) storage. The underlying data are mapped to RDF graphs and/or ontologies using languages such as the R2RML standard. SPARQL queries are rewritten to SQL queries following such mappings, which are evaluated over the underlying database; a more recent version rather translates SPARQL into an intermediate algebraic query that is subsequently optimised and translated into SQL [251]. Entailment for RDFS and OWL 2 QL are additionally supported through query rewriting techniques that expand the given query to capture solutions over entailments.

Quadstore [14] (2017) is a client-side RDF store that can be used with Node.js for in-browser management of RDF quads. The system also supports a variety of underlying storage options through the Level-down interface, such as LevelDB and RocksDB for persistent storage, and MemDown for in-memory storage. By default, indexes are generated for six quad permutations, namely SPOG, OGSP, GSPO, OSPG, POGS and GPOS, though these indexes are configurable by the user. SPARQL 1.1 queries and updates are supported.

AMBER [105] (2018) stores RDF graphs in a "multigraph" representation, where IRIs form nodes, whereas predicate–literal pairs form "attributes" on nodes. All nodes, predicates and attributes are dictionary encoded. AMBER then generates three indexes: the first stores the set of nodes for each attribute, the second stores vertex signatures that encode metadata about the triples where a given node is subject or object, and the third stores adjacency lists. Basic graph patterns are evaluated by classifying query nodes with degree greater than one as core nodes, and other nodes as satellite nodes. Core nodes are processed first, where candidates are produced for each query node based on the available indexes, recursively producing candidates for neighbors; the algorithm starts with the core query node with the most satellite nodes attached, or the highest degree. For each solution over the code nodes, each satellite node is then evaluated separately as they become disconnected once the core nodes are bound to constants.

TripleID-Q [49] (2018) is an RDF store that uses a compact representation called TripleID for RDF graphs such that query processing can be conducted on GPUs. The TripleID representation is based on a dictionary-encoded triple table. Rather than indexing the triple table, chunks of the table can

be loaded into GPUs, which, given a particular triple pattern, will scan the triple table in parallel looking for matching triples in the RDF graph. Other operators such as union, join, filter, distinct, etc., are then implemented on top of this GPU search; specifically, these operators are translated into functions that are executed in the GPU over the results of the search. RDFS entailment is further supported.

Jena-LTJ [99] (2019) extends the Jena TDB RDF store with the ability to perform worst-case optimal (wco) joins (see Section 6.3). Specifically, Jena TDB is extended with an algorithm similar to Leapfrog TrieJoin (LTJ), which is adapted from a relational setting for the RDF/SPARQL settings. The algorithm evaluates basic graph patterns variable-by-variable in a manner that ensures that the overall cost of enumerating all of the results is proportional to the number of results that it can return in the worst case. In order to reach wco guarantees, the three-order index of Jena TDB – based on B+trees – is extended to include all six orders. This ensures that for any triple pattern, the results for any individual variable can be read in sorted order directly from the index, which in turn enables efficient intersection of the results for individual variables across triple patterns. Thus Jena-LTJ uses twice the space of Jena TDB, but offers better query performance, particularly for basic graph patterns with cycles.

MAGiQ [109] (2019) is an RDF store that can use a variety of compressed sparse matrix/tensor representations for RDF graphs in order to translate basic graph patterns into linear algebra operations. These representations include compressed sparse column, doubly compressed sparse column, and coordinate list encodings of the graph as a matrix/tensor. Basic graph patterns are then translated into operations such as matrix multiplication, scalar multiplication, transposition, etc., over the associated matrices/tensor, which can be expressed in the languages provided by libraries such as GraphBLAS, Matlab, CombBLAS, and ultimately evaluated on CPUs and GPUs for hardware acceleration.

BMatrix [38] (2020) is a compact in-memory RDF store, where the RDF graph is first dictionary encoded and sorted by predicate. Two binary matrices are created: an $s \times n$ matrix called ST and an $o \times n$ matrix called OT, where s, o and n are the number of unique subjects, objects and triples respectively. The ST/OT matrix contains a 1 in index i, j if the subject/object of the jth triple corresponds to the ith term (or a 0 otherwise). Both matrices are indexed with k^2-trees, while a bit string of length n encodes predicate boundaries with a 1, i.e., in which columns of the matrix (denoting triples sorted or grouped by predicate) the predicate changes. These indexes are sufficient to cover all eight possible triple patterns. Further compression can be applied to the leaf matrices of the k^2.tree in order to trade space for time. The authors mention that joins can be supported in a similar fashion as used for RDFCSA and k^2-triples.

Tentris [31] (2020) is an in-memory RDF store wherein an RDF graph is viewed as a one-hot encoded 3-order tensor (equivalent to the 3-dimensional array used in BitMat [20]), which in turn is viewed as a trie of three levels for S, P and O. However, rather than storing tries for all permutations, a *hypertrie* is used with three levels. The leaves in the third level correspond to all possible combinations of two constants in a triple: for each triple (s, p, o), there exists a leaf that maps $(s, p, ?)$ to the set of all objects that replace $?$ in the graph (including o), with analogous leaves for $(?, p, o)$ and $(s, ?, o)$. The second level refers to single constants, where three nodes are present for $(s, ?, ?)$, $(?, p, ?)$, $(?, ?, o)$ such that $(s, ?, ?)$ will map to the leaves for $(s, p, ?)$ and $(s, ?, o)$, and likewise for $(?, p, ?)$ and $(?, ?, o)$. Finally, the top level – the root, representing zero constants – maps to all the second level elements. Basic graph patterns (with projection) are translated into tensor operations that can be evaluated on the hypertrie using a worst-case optimal join algorithm.

Ring [19] (2021) is an in-memory RDF store that uses FM-indexes (a text-indexing technique) in order to represent and index RDF graphs in a structure called a "ring". Specifically, a dictionary-encoded RDF graph is sorted lexicographically by subject-predicate-object; then the triples are concatenated to form a string $s_1 p_1 o_1 \ldots s_n p_n o_n$, where (s_i, p_i, o_1) indicates the ith (dictionary-encoded) triple in the order and $n = |G|$. A variant of a Burrows–Wheeler Transform is applied over this string, which allows for finding triples given any constant and position (or sequence of constants and positions), and for traversing to other elements of a triple in any direction. The result is a bidirectional circular index that covers all triple permutations with one index that encodes the graph and requires sub-linear space additional to the graph. For basic graph pattern queries, a variant of Leapfrog-Trie Join is implemented, offering worst-case optimal joins.

A.2 Distributed RDF Engines

We now survey distributed RDF stores. Table 5 summarizes the surveyed systems and the techniques they use. We further indicate the type of underlying storage used, where italicized entries refer to local stores. Some systems that appear in the following may have appeared before in the local discussion if they are commonly deployed in both settings.

YARS2 [94] (2007) is an RDF store based on similar principles to YARS (see local stores) but for a distributed environment. The index manager in YARS2 uses three indexes namely a quad index, keyword index, and a join index for evaluating queries. The quad indexes cover six permutations of quads. The keyword index is used for keyword lookups.

The join indexes help speed up query execution for common joins. The core index on quads is based on hashing the first element of the permutation, except in the case of predicates (e.g., for a POGS permutation), where hashing creates skew and leads to imbalance, and where random distribution is thus used. Indexed nested loop joins are used, with triple patterns being evaluated on one machine where possible (based on hashing), or otherwise on all machines in parallel (e.g., for constant predicates or keyword searches). Dynamic programming is used for join reordering in order to optimize the query.

Clustered TDB [173] (2008) is a distributed RDF store based on Jena TDB storage (a local system). The system is based on a master–slave architecture where the master receives and processes queries, and slaves index parts of the graph and can perform joins. Hash-based partitioning is used to allocate dictionary-encoded triples to individual slaves based on each position of the triple; more specifically, distributed SPO, POS and OSP index permutations are partitioned based on S, P and O, respectively. An exception list is used for very frequent predicates, which are partitioned by PO instead of P. Index-nested loop joins are supported and used to evaluate SPARQL basic graph patterns.

Virtuoso EE [69] (2008) is a local RDF store whose enterprise edition also offers support for indexing over a cluster of machines. Recalling that Virtuoso stores RDF graphs as a quads table in a custom relational database, the most recent version of Virtuoso offers three options for each table: partitioned, replicated or local. Partitioning is based on partition columns specified by the administrator, which are used for hash-based partitioning; partitions can also be replicated, if specified. Replication copies the full table to each machine, which can be used for query-based partitioning, or to store a global schema that is frequently accessed by queries. Local tables are only accessible to the individual machine, and are typically used for local configuration.

4store [91] (2009) stores quads over a cluster of machines, where subject-based hash partitioning is used. Three types of indexes are used in 4Store namely R, M, and P indexes. The R index is a hash table that dictionary encodes and stores meta-data about individual RDF terms (called "resources"). The M index is a hash table that maps graph names (called "models") to the corresponding triples in the named graph. The P Indexes consist of radix tries, with two for each predicate (similar to vertical partitioning): one for SOG order and another for OSG order. Joins are pushed, where possible, to individual machines. Join reordering uses cardinality estimations. SPARQL queries are supported.

Table 5: Categorization of distributed RDF Engines.

Storage: T = Triple Table, Q = Quad Table, V = Vertical Partitioning, P = Property table, G = Graph-based, E = Matrix/Tensor-based, M = Miscellaneous
Indexing: T = Triple, Q = Quad, E = Entity, P = Property, N = Path/Navigational, J = Join, S = Structural, M = Miscellaneous
Join P.: P = Pairwise, M = Multiway, W = Worst case optimal, L = Linear algebra
Query P.: R = Relational, N = Paths/Navigational, Q = Query rewriting
Partitioning: S = Statement (Triple/Quad)-based, G = Graph-based, Q = Query-based, R = Replication

Engine	Year	Storage							Indexing								Join P.				Query P.			Partitioning				Store
		T	Q	V	P	G	E	M	T	Q	E	P	N	J	S	M	P	M	W	L	R	N	Q	S	G	Q	R	
YARS2 [94]	2007		✓							✓							✓				✓			✓				Custom
Clustered TDB [173]	2008	✓							✓								✓				✓			✓				*Jena TDB*
Virtuoso EE [69]	2008	✓	✓							✓						✓	✓				✓	✓	✓	✓			✓	Custom
4store [91]	2009		✓							✓							✓				✓			✓				Custom
Blazegraph	2009	✓	✓						✓	✓							✓	✓			✓	✓		✓			✓	Custom
SHARD [195]	2009					✓											✓	✓			✓			✓				HDFS
Allegrograph	2010		✓							✓							✓				✓	✓					✓	Custom
GraphDB [125,33]	2010	✓	✓						✓	✓							✓				✓	✓					✓	Custom
AnzoGraph	2011		✓														✓				✓	✓		✓				Custom
CumulusRDF [131]	2011	✓	✓						✓	✓						✓	✓				✓			✓				Cassandra
H-RDF-3X [104]	2011	✓							✓								✓	✓			✓			✓			✓	*RDF-3X*
PigSPARQL [202]	2011	✓															✓	✓			✓		✓	✓				HDFS
Rapid+ [193]	2011			✓													✓	✓			✓		✓	✓				HDFS
AMADA [17]	2012	✓														✓	✓				✓			✓				SimpleDB
H2RDF(+) [178]	2012	✓							✓							✓	✓				✓			✓				HBase
Jena-HBase [121]	2012	✓		✓					✓							✓	✓				✓			✓				HBase
Rya [189]	2012	✓							✓							✓	✓				✓			✓				Accumulo
Sedge [254]	2012					✓											✓								✓	✓	✓	Pregel
chameleon-db [11]	2013					✓			✓					✓			✓				✓				✓	✓		Custom
D-SPARQ [160]	2013	✓							✓							✓	✓	✓			✓						✓	MongoDB
EAGRE [261]	2013	✓				✓			✓		✓						✓	✓			✓				✓			HDFS
MR-RDF [64]	2013	✓							✓								✓	✓			✓				✓			*RDF-3X*
SHAPE [134]	2013	✓							✓								✓	✓						✓				*RDF-3X*
Trinity.RDF [258]	2013	✓							✓		✓						✓				✓				✓			Trinity
TripleRush [220]	2013	✓							✓								✓							✓				Signal/Collect
WARP [103]	2013	✓							✓								✓	✓								✓	✓	*RDF-3X*
AKZ14 [6]	2014	✓			✓				✓							✓	✓	✓			✓		✓	✓	✓			HBase
Partout [73]	2014	✓							✓								✓				✓						✓	*RDF-3X*
P-LUPOSDATE [79]	2014	✓							✓							✓	✓				✓		✓	✓				HBase
RDF-3X-MPI [54]	2014	✓							✓								✓							✓	✓		✓	*RDF-3X*
Sempala [203]	2014				✓											✓	✓	✓			✓			✓				HDFS
SemStore [244]	2014			✓		✓			✓								✓	✓							✓			*TripleBit*
SparkRDF [51]	2014			✓					✓					✓			✓				✓			✓				HDFS/Spark
TriAD [86]	2014	✓							✓					✓	✓		✓				✓				✓			Custom
CK15 [53]	2015			✓					✓					✓			✓				✓				✓	✓		Custom
CliqueSquare [75]	2015	✓							✓								✓	✓			✓						✓	HDFS
DREAM [88]	2015	✓							✓								✓					✓					✓	*RDF-3X*
AdPart [89]	2016	✓							✓								✓				✓						✓	Custom
DiploCloud [248]	2016					✓			✓		✓						✓				✓				✓	✓		Custom
Dydra [15,14]	2016			✓					✓								✓				✓						✓	Custom
gStore-D [182]	2016					✓			✓		✓							✓			✓				✓			gStore
Halyard [213]	2016	✓	✓						✓	✓							✓				✓						✓	HBase
JARS [192]	2016	✓							✓								✓				✓			✓				Custom
S2RDF [204]	2016			✓												✓	✓				✓		✓	✓				HDFS
S2X [201]	2016	✓															✓				✓		✓	✓	✓			Spark
SPARQLGX [76]	2016			✓					✓								✓				✓			✓				HDFS
Wukong [211]	2016	✓							✓							✓	✓				✓							DrTM-KV
CM-Well [30]	2017	✓													✓		✓				✓	✓		✓				Cassandra/ElasticS.
Koral [112]	2017	✓							✓								✓							✓	✓			Custom
MarkLogic	2017	✓							✓								✓				✓	✓		✓				Custom
SANSA [136]	2017	✓		✓													✓						✓	✓				HDFS
Spartex [5]	2017					✓			✓									✓						✓				GSP/Custom
Stylus [96]	2017					✓			✓		✓	✓						✓			✓			✓			✓	Trinity
Neptune [27]	2018		✓							✓							✓				✓	✓					✓	Custom
PRoST [60]	2018			✓	✓											✓	✓	✓					✓	✓				HDFS
RDFox-D [188]	2018	✓							✓								✓								✓			*RDFox*
WORQ [146]	2018			✓										✓			✓								✓			Spark
Wukong+G [235]	2018	✓				✓			✓								✓								✓			*Wukong*
Akutan	2019	✓							✓								✓				✓	✓		✓			✓	RocksDB
DiStRDF [239]	2019	✓		✓												✓	✓				✓			✓				HDFS
gStore-D2 [181]	2019					✓			✓						✓		✓	✓			✓				✓	✓	✓	Custom
Leon [83]	2019	✓							✓			✓					✓				✓						✓	Custom
SPT+VP [95]	2019			✓	✓				✓								✓				✓		✓	✓				Spark
StarMR [236]	2019					✓			✓								✓	✓						✓				HDFS
DISE [107]	2020	✓					✓		✓								✓						✓	✓				Spark
DP2RPQ [237]	2020					✓																✓	✓	✓				Spark
Triag [162]	2020					✓									✓		✓	✓					✓		✓			Spark
WISE [84]	2020	✓							✓					✓			✓	✓					✓	✓	✓			Leon
gSmart [52]	2021						✓		✓									✓		✓	✓			✓		✓		Custom

Blazegraph [224] (2009), discussed previously as a local store, also features partitioning in the form of key-range shards that allow for partitioning B+tree indexes, potentially across multiple machines. An alternative replication cluster is supported that indexes the full RDF graph or SPARQL dataset on each machine, allowing queries to be evaluated entirely on each machine without network communication.

SHARD [195] (2009) is a distributed, Hadoop-based RDF store. It stores an RDF graph in flat files on HDFS such that each line presents all the triples associated with a given subject resource of the RDF triple, which can be seen as an adjacency list. The graph is hash partitioned, so that every partition contains a distinct set of triples. As the focus is on batch processing of joins, rather than evaluating queries in real-time, there is no specific indexing employed in SHARD. Query execution is performed through MapReduce iterations: first, it collects the results for the subqueries, which are joined and finally filtered according to bound variables and to remove redundant (duplicate) results.

AllegroGraph (2010), discussed previously as a local store, offers a distributed version where data are horizontally partitioned into shards, which are indexed locally on each machine per the local version. Alongside these shards, "knowledge bases" can be stored, consisting of triples that are often accessed by all shards (e.g., schema or other high level data), such that queries can be evaluated (in a federated manner) over one shard, and potentially several knowledge bases.

GraphDB [125,33] (2010), also a local store, offers an enterprise edition that can store RDF graphs on a cluster of machines using a master–slave architecture, where each cluster has at least one master node that manages one or more worker nodes, each replicating the full database copy, thus allowing for queries to be evaluated in full on any machine. Updates are coordinated through the master.

AnzoGraph [15] (2011) is an in-memory, massively parallel processing (MPP) RDF store based on a master–slave architecture. The system indexes named graphs, where partitioning and replication are also organized by named graphs. By default, all triples involving a particular term are added into a named graph for that term. A dictionary is provided to map terms to named graphs. The query is issued at a master node, which features a query planner that decides the type of join (hash or merge joins are supported) or aggregation needed. Individual operations are then processed over the slaves in parallel, generating a stream of intermediate results that are combined on the master.

---

[15] https://docs.cambridgesemantics.com/anzograph/userdoc/features.htm

CumulusRDF [131] (2011) works on top of Apache Cassandra: a distributed key-value store with support for tabular data. Three triple permutations – SPO, POS, OSP – and one quad permutation – GSPO – are considered. A natural idea would be to index the first element as the row key (e.g., S for SPO), the second (e.g., P) as the column key, and the third (e.g., O) as the cell value, but this would not work in multi-valued cases as columns are unique per row. Two other data storage layouts are thus proposed. Taking SPO, the "hierarchical layout" stores S as the row key (hashed and used for partitioning), P as the supercolumn key (sorted), O as the column key (sorted), with the cell left blank. An alternative that outperforms the hierarchical layout is a "flat layout", where for SPO, S remains the row key, but PO is concatenated as the column key, and the cell is left blank. In the POS permutation, the P row key may create a massive row; hence PO is rather used as the row key, with P being indexed separately. Join and query processing is enabled though Sesame.

H-RDF-3X [104] (2011) is a Hadoop-based RDF store that uses RDF-3X on a cluster of machines. A graph-based partitioning (using the METIS software package) is used to distribute triples among multiple worker nodes. It also employs a k-hop guarantee, which involves replicating nodes and edges that are k hops away from a given partition, thus increasing the locality of processing possible, and reducing communication costs. Local joins are optimized and evaluated on individual machines by RDF-3X, while joins across machines are evaluated using Hadoop. The use of Hadoop – which involves expensive coordination across machines, and heavy use of the disk – is minimized by leveraging the k-hop guarantee and other heuristics.

PigSPARQL [202] (2011) is a Hadoop-based RDF store that uses a vertical partitioning strategy. Data are stored on HDFS without indexes, and thus the focus is on batch processing. SPARQL queries are translated into PigLatin: an SQL-inspired scripting language that can be compiled into Hadoop tasks by the Pig framework. The Jena ARQ library is used to parse SPARQL queries into an algebra tree, where optimizations for filters and selectivity-based join reordering are applied. The tree is traversed in a bottom-up manner to generate PigLatin expressions for every SPARQL algebra operator. The resulting PigLatin script is then translated to – and run as – MapReduce jobs on Hadoop.

Rapid+ [193] (2011) is a Hadoop-based system that uses a vertical partitioning strategy for storing RDF data. Without indexing, the system targets batch processing. Specifically, Pig is used to generate and access tables under a vertical partitioning strategy. In order to translate SPARQL queries into PigLatin scripts, user-defined-functions are implemented that allow for optimizing common operations, such as loading

and filtering in one step. Other optimizations include support for star joins using grouping, and a look-ahead heuristic that reduces and prepares intermediate results for operations that follow; both aim to reduce the number of Hadoop tasks needed to evaluate a query.

AMADA [17] (2012) is an RDF store based on the Amazon Web Services (AWS) cloud infrastructure. Indexes for the RDF graph are built using Amazon SimpleDB: a key-value storage solution that supports a subset of SQL. SimpleDB offers several indexing strategies, where "attribute indexing" can be used to create three indexes for the three elements of a triple. In AMADA, a query is submitted to a query processing module running on EC2, which in turn evaluates triple patterns using the SimpleDB-based indexes.

H2RDF(+) [178, 177] (2012) stores RDF graphs using the HBase distributed tabular NoSQL store. Three triple permutations (SPO, POS, and OSP) are created over HBase tables in the form of key-value pairs. A join executor module creates the query plan, which decides between the execution of joins in a centralized (local) and distributed (Hadoop-based) manner. It further reorders joins according to selectivity statistics. Multiway (sort-)merge joins are run in Hadoop.

Jena-HBase [121] (2012) (also known as *HBase-RDF*[16]) is a distributed RDF store using HBase as its back-end. Jena-HBase supports three basic storage layouts for RDF graphs in HBase namely "simple": three triple tables, the first indexed and partitioned by S, the second by P, the third by O; "vertical partitioning": two tables for each predicate, one indexed by S, the other by O; "indexed": six triple tables covering all permutations of a triple. Hybrid layouts are also proposed that combine the basic layouts, and are shown to offer better query times at the cost of additional space. Jena is used to process joins and queries.

Rya [189] (2012) is a distributed RDF store that employs Accumulo – a key-value and tabular store – as its back-end. However, it can also use other NoSQL stores as its storage component. Rya stores three index permutations namely SPO, POS, and OSP. Query processing is based on RDF4J, with index-nested loop joins being evaluated in a MapReduce fashion. The count of the distinct subjects, predicates, and objects is maintained and used during join reordering and query optimization.

Sedge [254] (2012) is an RDF store based on Pregel: a distributed (vertex-centric) graph processing framework. Pregel typically assumes a strict partition of the nodes in a graph,

where Sedge relaxes this assumption to permit nodes to coexist in multiple partitions. A complementary graph partitioning approach is proposed involving two graph partitionings, where the cross-partition edges of one are contained within a partition of the other, reducing cross-partition joins. Workload-aware query-based partitioning is also proposed, where commonly accessed partitions and frequently-queried cross-partition "hotspots" are replicated. The store is implemented over Pregel, where indexes are built to map partitions to their workloads and to their replicas, and to map nodes to their primary partitions.

chameleon-db [11] (2013) is a distributed RDF store using custom graph-based storage. Partitioning is graph-based and is informed by the queries processed, which may lead to dynamic repartitioning to optimize for the workload being observed. An incremental indexing technique – using a decision tree – is used to keep track of partitions relevant to queries. It also uses a hash-table to index the nodes in each partition, and a range-index to keep track of the minimum and maximum values for literals of each distinct predicate in each partition. The evaluation of basic graph patterns is delegated to a subgraph matching algorithm over individual partitions, whose results are then combined in a query processor per the standard relational algebra. Optimizations involve rewriting rules that preserve the equivalence of the query but reduce intermediate results.

D-SPARQ [160] (2013) is a distributed RDF store built on top of MongoDB: a NoSQL store for JSON-like documents. D-SPARQ partitions the RDF graph by subject. Partial data replication is used whereby selected triples are replicated across partitions to increase parallelism when executing (sub-)queries. Indexes are built for SP and PO permutations. D-SPARQ optimizes multiway S–S (star) joins, taking advantage of the locality offered by the S-based partitioning; selectivity estimates are used to reorder joins.

EAGRE [261] (2013) stores RDF data on HDFS, where data are pre-processed using Hadoop to extract entities and their classes, thereafter applying graph-based data partitioning using METIS. For each entity class, EAGRE adopts a space-filling curve technique (see Section 5.2): an in-memory index structure that is used to index high-dimensional data, and more specifically in this case, to decide where the data for a given entity should be stored. Joins are pushed to individual nodes where possible, with multiway joins between nodes being evaluated using Hadoop. A strategy similar to a distributed form of sideways-information-passing is used to reduce network traffic, where nodes share information about the possible ranges of constants matching individual variables, filtering intermediate results outside those ranges before they are sent over the network.

---

[16] https://github.com/castagna/hbase-rdf

MR-RDF [64] (2013) is a distributed RDF store that uses RDF-3X for local storage, and Hadoop for join processing. A partition is generated for each characteristic set, where a triple is added to the partition for the characteristic set of its subject. Given that this may give rise to a large number of partitions, similar characteristic sets are clustered together to form larger partitions corresponding to the number of machines available. The larger partition is then described by the union of the characteristic sets it contains, which can be used for matching star joins (with constant predicates) to partitions. Star joins are evaluated locally by RDF-3X, and their results are joined over Hadoop.

SHAPE [134] (2013) uses RDF-3X to store and index RDF triples on a distributed cluster of machines. Triples are partitioned using a semantic hash partitioning scheme that is based on the IRI prefix hierarchy: triples with the same subject or object prefixes are identified and are placed in the same partition. The intuition is that such triples are more likely to be queried together. A distributed query execution planner coordinates the intermediate results from different nodes, which are joined using Hadoop.

Trinity.RDF [258] (2013) is an RDF store implemented on top of Trinity: a distributed memory-based key-value storage system. A graph-based storage scheme is used, where an inward and outward adjacency list is indexed for each node. Hash-based partitioning is then applied on each node such that the adjacency lists for a given node can be retrieved from a single machine; however, nodes with a number of triples/edges exceeding a threshold may have their adjacency lists further partitioned. Aside from sorting adjacency lists, a global predicate index is also generated, covering the POS and PSO triple permutations. Queries are processed through graph exploration, with dynamic programming over cardinality estimates used to choose a query plan.

TripleRush [220] (2013) is based on the Signal/Collect distributed graph processing framework [219]. In this framework, TripleRush considers an in-memory graph with three types of nodes. Triple nodes embed an RDF triple with its subject, predicate, and object. Index nodes embed a triple pattern. Query nodes coordinate the query execution. The index graph is formed by index and triple nodes, which are linked based on matches. A query execution is initialized when a query node is added to a TripleRush graph. The query vertex emits a query particle (a message) which is routed by the Signal/Collect framework to index nodes for matching. Partitioning of triples and triple patterns is based on the order S, O, P, where the first constant in this order is used for hash-based partitioning. Later extensions explored workload-aware query-based partitioning methods [226].

WARP [103] (2013) uses RDF-3X to store triples in partitions among a cluster of machines. Like H-RDF-3X, graph-based partitioning is applied along with a replication strategy for k-hop guarantees. Unlike H-RDF-3X, WARP proposes a query-based, workload-aware partitioning, whereby the value of k is kept low, and selective replication is used to provide guarantees specifically with respect to the queries of the workload, reducing storage overheads. Sub-queries that can be evaluated on one node are identified and evaluated locally, with custom merge joins (rather than Hadoop, as in the case of H-RDF-3X) used across nodes. Joins are reordered to minimize the number of single-node subqueries.

AKZ14 [6] (2014) is a distributed RDF store based on the HBase tabular store. A property table storage scheme is implemented over HBase, which is built based on clustering entities with similar properties. A secondary triple table is used for multi-valued properties and (infrequent) properties that do not appear in the clusters. Property tables are used to solve subject-based star joins, with other joins being evaluated over Hadoop by translating SPARQL queries to Hive (an SQL-like language for Hadoop). Metadata for the HBase tables are stored in a relational database (MySQL).

Partout [73] (2014) is a distributed store that uses RDF-3X for underlying storage on each machine. The RDF graph is partitioned using a workload-aware query-based partitioning technique, aiming to group together triples that are likely to be queried together. Each partition is indexed using standard RDF-3X indexing. The SPARQL query is issued to a query processing master, which uses RDF-3X to generate a suitable query plan according to a global statistics file. The local execution plan of RDF-3X is transformed into a distributed plan, which is then refined by a distributed cost model that assigns subqueries to partitions. This query plan is executed by slave machines in parallel, whose results are combined in the master.

P-LUPOSDATE [79] (2014) is a distributed RDF store built on HBase. Triples are distributed according to six triple permutations – partitioning on S, P, O, SP, SO, PO – enabling lookups for any triple pattern. In order to reduce network communication, Bloom filters are pre-computed for each individual variable of each triple pattern with at least one constant and one variable that produces some result; e.g., for SP, a Bloom filter is generated encoding the objects of each subject–predicate pair; for S, a Bloom filter is generated for each subject encoding its predicates, and optionally, another Bloom filter is generated for its objects. These Bloom filters are sent over the network in order to compute approximate semi-join reductions, i.e., to filter incompatible results before they are sent over the network. SPARQL (1.0) queries are evaluated by translating them to PigLatin, which are compiled into Hadoop jobs by Pig.

RDF-3X-MPI [54] (2014) is a distributed RDF store build on top of RDF-3X and a Message Passing Interface (MPI). After dictionary-encoding the triples, they are initially partitioned based on hashes on graph nodes, where the partitions are extended to ensure an n-hop guarantee: i.e., that any node reachable in n-hops from a node assigned to that partition will also be available on the same partition. The partitions are stored in RDF-3X on each machine, and basic graph patterns are evaluated independently on each partition (it is assumed that the value of n is sufficient to enable this, with other queries left for future work).

Sempala [203] (2014) stores RDF triples in a distributed setting, using the columnar Parquet format for HDFS that supports queries for specific columns of a given row (without having to read the full row). In this sense, Parquet is designed for supporting a single, wide (potentially sparse) table and thus Sempala uses a single "unified property table" for storing RDF triples with their original string values; multi-valued properties are stored using additional rows that correspond to a Cartesian product of all values for the properties of the entity. SPARQL queries are translated into SQL, which is executed over the unified property table using Apache Impala: a massively parallel processing (MPP) SQL engine that runs over data stored in HDFS.

SemStore [244] (2014) is a distributed RDF store with a master–slave architecture. A custom form of graph partitioning is used to localize the evaluation of subqueries of particular patterns – star, chain, tree, or cycle – that form the most frequent elements of basic graph patterns. A k-means partitioning algorithm is used to assign related instances of patterns to a particular machine, further increasing locality. The master creates a global bitmap index over the partitions and collects global cardinality-based statistics. Slave nodes use the TripleBit local RDF engine for storage, indexing and query processing. The master node then generates the query plan using dynamic programming and global cardinality statistics, pushing joins (subqueries) to individual slave nodes where possible.

SparkRDF [51] (2014) is a Spark-based RDF engine that distributes the graph into subgraphs using vertical partitioning, adding tables for classes as well as properties. SparkRDF then creates indexes over the class and property tables, and further indexes class–property, property–class, and class–property-class joins. These indexes are loaded into an in-memory data structure in Spark (a specialized RDD) that implements query processing functionalities such as joins, filters, etc. Class information is used to filter possible results for individual variables, where a greedy selectivity-based

strategy is used for reordering joins. Joins themselves are evaluated in a MapReduce fashion.

TrIAD [86] (2014) is a in-memory distributed RDF store based on a master–slave architecture. The master maintains a dictionary of terms, a graph summary that allows for pruning intermediate results, as well as global cardinality-based statistics that allow for query planning. The graph summary is a quotient graph using METIS' graph partitioning: each partition forms a supernode, and labeled edges between supernodes denote triples that connect nodes in different partitions; the graph summary is indexed in two permutations: PSO and POS. The triples for each partition are stored on a slave; triples connecting two partitions are stored on both slaves. Each slave indexes their subgraph in all six triple permutations. Given a basic graph pattern, the graph summary is used to identify relevant partitions, which are shared with the slaves and used to prune results; dynamic programming uses the global statistics to optimize the query plan. Alongside distributed hash and merge joins, an asynchronous join algorithm using message passing is implemented.

CK15 [53] (2015) is a distributed in-memory RDF store that combines two types of partitioning: triple-based partitioning and query-based partitioning. The graph is initially divided over the machines into equal-size chunks and dictionary-encoded in a distributed manner (using hash-based partitioning of terms). The encoded triples on each machine are then stored using a vertical partitioning scheme, where each table is indexed by hashing on subject, and on object, providing P $\rightarrow$ SO, PS $\rightarrow$ O and PO $\rightarrow$ S lookups. Parallel hash joins are proposed. Secondary indexes are then used to cache intermediate results received from other machines while processing queries, such that they can be reused for future queries. These secondary indexes can also be used for computing semi-join reductions on individual machines, thus reducing network traffic.

CliqueSquare [75] (2015) is a Hadoop-based RDF engine used to store and process massive RDF graphs. It stores RDF data in a vertical partitioning scheme using semantic hash partitioning, with the objective of enabling co-located or partitioned joins that can be evaluated in the map phase of the MapReduce paradigm. CliqueSquare also maintains three replicas for fast query processing and increased data locality. In order to evaluate SPARQL queries, CliqueSquare uses a clique-based algorithm, which works in an iterative way to identify cliques in a query–variable graphs and to collapse them by evaluating joins on the common variables of each clique. The process will then terminate when the query–variable graph consists of only one node.

DREAM [88] (2015) is a distributed store using RDF-3X for its underlying storage and indexing. The entire RDF graph is replicated on every machine, with standard RDF-3X indexing and query processing being applied locally. To reduce communication, dictionary-encoded terms are shared within the cluster. In the query execution phase, the SPARQL query is initially represented as a directed graph, which is divided into multiple subqueries to be evaluated by different machines. The results of subqueries are combined using hash joins and eventually dictionary-decoded.

AdPart [89] (2016) is a distributed in-memory RDF store following a master–slave architecture. The master initially performs a hash-based partitioning based on the subjects of triples. The slave stores the corresponding triples using an in-memory data structure. Within each slave, AdPart indexes triples by predicate, predicate–subject, and predicate–object. Each slave machine also maintains a replica index that incrementally replicates data accessed by many queries; details of this replication are further indexed by the master machine. Query planning then tries to push joins locally to slaves (hash joins are used locally), falling back to distributed semi-joins when not possible. Join reordering then takes communication costs and cardinalities into account.

DiploCloud [248] (2016) is a distributed version of the local RDF store dipLODocus. The store follows a master–slave architecture, where slaves store "molecules" (see the previous discussion on dipLODocus). The master provides indexes for a dictionary, for the class hierarchy (used for inference), as well as an index that maps the individual values of properties selected by the administrator to their molecule. Each slave stores the molecule subgraphs, along with an index mapping nodes to molecules, and classes to nodes. Query processing pushes joins where possible to individual slaves; if intermediate results are few, the master combines results, or otherwise a distributed hash join is employed. Molecules can be defined as a k-hop subgraph around the root node, based on input from an administrator, or based on a given workload of queries.

Dydra [15, 14] (2016) is an RDF store that can leverage both local and remote storage, and provides support for versioned RDF graphs. In terms of local storage, RDF data are dictionary encoded and indexed in six permutations of quad tables – namely GSPO, GPOS, GOSP, SPOG, POSG, OPSG – using on-disk B+trees. These B+trees offer support for both static and streaming data, and further capture information about revisions, enabling versioned queries and other RDF archival features. Support for replication through convergent replicated data types (CvRDTs) is also proposed [14]. A SPARQL query processor is layered on top of storage, providing support for SPARQL 1.1 queries and updates. Dydra further offers a multi-tenant cloud-based storage service.

gStore-D [182] (2016) is a distributed RDF store that uses a variation of the local gStore RDF engine for local storage and indexing (it uses adjacency lists and vertex signatures, as discussed previously). A graph partitioning algorithm is applied over the RDF graph, with the subgraphs induced by each partition being assigned to individual machines. Edges that connect distinct partitions are indexed within the subgraphs of both partitions. Where possible, joins are then pushed to the individual subgraphs. Joins across subgraphs can be evaluated in a central (i.e., by a single master) or distributed (i.e., by several slaves) manner. Support for the relational features of SPARQL (1.0) is described.

Halyard [213] (2016) is a distributed RDF store that combines RDF4J with underlying HBase storage. MapReduce (Hadoop) is used to perform a bulk load of RDF quads into HBase tables. Sorted hash-based indexes are built in six permutations – SPO, POS, OSP, GSPO, GPOS, GOSP – enabling efficient lookups for any quad pattern. HBase is connected with RDF4J as a Storage and Inferencing Layer (SAIL), enabling support for SPARQL 1.1. Nested-loop joins are used, and various optimizations are implemented for distributed evaluation, including push-based physical operators, priority queues for parallelizing operations, etc. Inferencing is supported by materializing entailments and bulk-loading them into HBase using the MapReduce framework.

JARS [192] (2016) is a distributed RDF store that combines triple-based and query-based partitioning. The graph is partitioned by hashing on subject, and hashing on object, constructing two distributed triple tables. The subject-hashed table is indexed on the POS, PSO, OSP and SPO permutations, while the object-hashed table is indexed on POS, PSO, SOP and OPS. Specifically, by hashing each triple on subject and object, the data for S–S, O–O and S–O are on one machine; the permutations then allow for such joins to be supported as merge joins on each machine. Basic graph patterns are then decomposed into subqueries answerable on a single machine, with a distributed hash join applied over the results. Jena ARQ is used to support SPARQL.

S2RDF [204] (2016) is a distributed RDF store based on HDFS (with Parquet). The storage scheme is based on an extended version of vertical partitioning with semi-join reductions (see Section 4.3). This scheme has a high space overhead, but ensures that only data useful for a particular (pairwise) join will be communicated over the network. In order to reduce the overhead, semi-join tables are not stored in cases where the selectivity of the join is high; in other words, semi-join tables are stored only when many triples are filtered by the semi-join (the authors propose a threshold

of 0.25, meaning that at least 75% of the triples must be filtered by the semi-join for the table to be included). SPARQL queries are optimized with cardinality-based join reordering, and then translated into SQL and evaluated using Spark.

S2X [201] (2016) runs SPARQL queries over RDF graphs using GraphX: a distributed graph processing framework built on the top of Spark. The triples are stored in-memory on different slave machines with Spark (RDDs), applying a hash-based partitioning on subject and objects (per GraphX's default partitioner). S2X does not maintain any custom indexing. For SPARQL query processing, graph pattern matching is combined with relational operators (implemented in the Spark API) to produce solution mappings.

SPARQLGX [76] (2016) stores RDF data on HDFS per a vertical partitioning scheme. A separate file is created for each unique predicate in the RDF graph, with each file containing the subjects and objects of that triple. No indexes are provided, and thus the system is intended for running joins in batch-mode. SPARQL queries are first optimized by applying a greedy join reordering based on cardinality and selectivity statistics; the query plan is then translated into Scala code, which is then directly executed by Spark.

Wukong [211] (2016) stores RDF graphs in DrTM-KV: a distributed key–value store using "remote direct memory access" (RDMA), which enables machines to access the main memory of another machine in the same cluster while bypassing the remote CPU and OS kernel. Within this store, Wukong maintains three kinds of indexes: a node index that maps subjects or (non-class) objects to their corresponding triples; a predicate index, which returns all subjects and objects of triples with a given predicate; and a type index, which returns the class(es) to which a node belongs. Hash-based partitioning is used for the node index, while predicate and type indexes are split and replicated to improve balancing. A graph-traversal mechanism is used to evaluate basic graph patterns, where solutions are incrementally extended or pruned. For queries involving fewer data, the data are fetched from each machine on the cluster and joined centrally; for queries involving more data, subqueries are pushed in parallel to individual machines. A work-stealing mechanism is employed to provide better load balancing while processing queries.

CM-Well [30] (2017) is a distributed RDF store developed by Thomson Reuters (Refinitiv) that combines Cassandra and ElasticSearch for underlying storage, further using Akka and Kafka for coordination and communication. RDF triples are grouped by node (subject), and stored in Cassandra, with inverted indexes for each node indexed by ElasticSearch. In order to process queries, two modes are considered. In sub-graph mode, candidate nodes are identified using ElasticSearch, and their associated triples are loaded in Jena in order to process. In full-graph mode, queries are translated directly to ElasticSearch operators.

Koral [112] (2017) is a distributed RDF store based on a modular master–slave architecture that supports various options for each component of the system. Among these alternatives, various triple-based and graph-based partitioning schemes are supported. In order to evaluate basic graph patterns, joins are processed in an analogous way to TrIAD, using asynchronous execution, which makes the join processing strategy independent of the partitioning chosen. The overall focus of the system is to be able to quickly evaluate different alternatives for individual components – particularly partitioning strategies – in a distributed RDF store.

MarkLogic [17] (2017) is a multi-model distributed store with support for XML, JSON and RDF. Originally focusing on the storage and querying of XML documents, support for RDF and SPARQL was added in 2017 with the release of MarkLogic Server v.7.0, with SPARQL 1.1 support added in v.8.0. MarkLogic stores dictionary-encoded RDF triples in 4 kilobyte blocks, over which an LRU cache is implemented. Indexes are built for three permutations: PSO, SOP and OPS. SPARQL 1.1 is supported, as well as rule-based inferencing implemented using backward chaining. Distribution is enabled through "evaluator nodes" that perform query processing, and "data nodes" that store and index data.

SANSA [136] (2017) is a Spark-based distributed RDF store. RDF data are stored on HDFS, where triple-based partitioning – such as predicate-based vertical partitioning – is employed. Queries are transformed into Spark (or Flink) programs, using Sparklify [214]: a query engine for SPARQL-to-SQL translations, which can be run on Spark. SANSA is part of a larger stack that supports RDF-based inferencing and machine learning in a distributed environment.

Spartex [5] (2017) is a distributed RDF store with analytical capabilities. An extension of SPARQL queries is proposed with user-defined procedures for analytics (e.g., PageRank), among other features. The system is built on top of GPS: an open-source implementation of Pregel's distributed, vertex-centric graph processing framework. A master–slave architecture is employed. The master is responsible for query planning and manages global statistics. The RDF graph is partitioned among its slaves; namely each (subject/object) node and its incident edges (triples) is assigned to a slave. Each slave stores and indexes its subgraph in-memory using PS and PO permutations. Basic graph patterns are then evaluated using graph traversals in GSP, with nodes (vertexes)

---

[17] https://docs.marklogic.com/guide/semantics

sharing intermediate results as messages, which are joined with local data. Optimizations are based on minimizing duplicate traversals involving (non-Eulerian) cycles, as well as traditional cardinality estimates.

Stylus [96] (2017) is a distributed RDF store using Trinity: a graph engine based on an in-memory key–value store. Terms of the RDF graph are dictionary encoded. Each subject and object node is associated with a dictionary identifier and its characteristic set. A sorted adjacency list (for inward and outward edges) is then stored for each node that also encodes an identifier for the characteristic set of the node. Schema-level indexes for characteristic sets are replicated on each machine. Hash-based partitioning is employed on the data level. Indexes are used to efficiently find characteristic sets that contain a given set of properties, as well as to evaluate common triple patterns. Given a basic graph pattern, the characteristic sets are used to prune intermediate results on star joins, where candidates are kept for each variable node in the query. Cardinality-based join reordering is applied. Relational features of SPARQL (1.0) are supported.

Neptune [27] (2018) is an RDF store that is hosted as a service on Amazon's S3 cloud storage infrastructure. Neptune stores SPARQL datasets in the form of quads with three index permutations: SPOG, POGS and GSPO; this is sufficient to cover 9 out of 16 possible quad patterns. Neptune makes use of cardinality estimations and static analysis to rewrite queries. Partitioning is not supported, where Neptune rather offers up to 16 replicas of the full graph to increase query throughput; a primary replica is nominated to receive and coordinate updates. Graphs in Neptune can be queried (and processed) through the SPARQL 1.1, Apache TinkerPop and Gremlin languages.

PRoST [60] (2018) is a distributed RDF store using HDFS storage and Spark query processing. The storage scheme uses a combination of vertical partitioning and property tables that aims to leverage the strengths and minimize the weaknesses of both schemes. Like Sempala, the property table is stored in the column-wise Parquet format; multi-valued properties are supported by adding lists of values. Star joins on a common subject variable are evaluated on the property table, while other patterns and joins are addressed with the vertical partitioning tables. Selectivity-based heuristics are used to reorder joins. Queries are then rewritten into SQL for execution with Spark.

RDFox-D [188] (2018) is a distributed in-memory RDF store based on RDFox that uses distributed index nested loop joins. A global index is built mapping the terms of the RDF graph to the partitions it appears in. The graph is partitioned by a weighted graph-based partitioning scheme, where nodes are weighted by the number of triples they appear in as subject. The partitioning minimizes cross-partition edges while balancing the sum of the node weights in each partition. Triples with a subject in the same partition are sent to the same machine; the weights used for partitioning then help to ensure more even balancing. Joins are evaluated in a pairwise manner, where each machine extends solutions asynchronously, without central coordination, based on its partition of the graph; it then sends the extended partial solution to the machines that can potentially extend it further (based on the global index). Termination occurs when all partial solutions have been forwarded. Various optimizations are discussed. Joins are reordered based on cardinalities.

WORQ [146] (2018) is a distributed RDF store that uses a workload-aware approach to partition data. In order to reduce the number of intermediate results, Bloom filters are used to index the constants matching the variable of a given triple pattern, which are shared and used to filter results for that variable elsewhere. Bloom filters provide an approximate membership function (i.e., they may yield false positives), and thus a distributed join algorithm must be applied over the (reduced) intermediate results. Further Bloom filters can be computed for multiway joins, analogous to an approximate form of semi-join reduction (as used by S2RDF). These reductions can be cached for later re-use, where they are partitioned across machines based on the join element. WORQ is implemented over Spark.

Wukong+G [235] (2018) extends the distributed RDF store Wukong [211] in order to additionally exploit GPUs (as well as CPUs) for processing queries in a distributed environment. One of the main design emphases of the system is to ensure that large RDF graphs can be processed efficiently on GPUs by ensuring effective use of the memory available, noting in particular that the local memory of GPUs has a much higher bandwidth for reading data into the GPU's cores, but a much lower capacity than typical for CPU RAM. Wukong+G thus employs a range of memory-oriented optimizations involving prefetching, pipelining, swapping, etc., to ensure efficient memory access when processing queries on the GPU. A graph partitioning algorithm is further employed to distribute storage, where lower-cost queries are processed on CPU (as per Wukong), but heavier loads are delegated to GPUs, where (like Wukong) efficient communication is implemented using RDMA primitives, allowing more direct access to remote CPU and GPU memory.

Akutan [18] (2019) (formerly known as *Beam*) is a distributed RDF store developed by eBay. Triple storage is implemented on top of RocksDB, with indexes provided on SP → O and

---

[18] https://tech.ebayinc.com/engineering/akutan-a-distributed-knowledge-graph-store/

OP → S. Triples are additionally associated with triple identifiers. Transactional logging is implemented using Apache Kafka, which coordinates read and write requests across machines. A SPARQL(-like) query processor is then layered on top of the underlying storage layer, which includes an optimizer that leverages statistics about the data to reorder joins. Hash joins and nested-loop joins are supported, and selected, as appropriate, by the query planner. Queries are then processed in streams and/or batches. A limited form of inference based on transitive closure is also supported.

DiStRDF [239] (2019) is a massively parallel processing (MPP) RDF store based on Spark with support for spatio-temporal queries. A special dictionary-encoding mechanism is used where the identifier concatenates a bit-string for spatial information, a bit-string for temporal information, and a final bit-string to ensure that the overall identifier is unique. Thus spatial and temporal processing can be applied directly over the identifiers. Storage based on both a triple table and property tables is supported, where range-based partitioning is applied to the triples (based on the spatio-temporal information). Data is stored on HDFS in CSV or Parquet formats. Query processing is implemented in Spark. Distributed hash joins and sort–merge joins are supported; selections and projections are also supported. Three types of query plans are proposed that apply RDF-based selections, spatio-temporal selections and joins in different orders.

gStore-D2 [181] (2019) is a distributed RDF store using workload-aware graph partitioning methods. Frequently accessed (subgraph) patterns are mined from the workload, where all subjects and objects are mapped to variables. Subgraphs that instantiate these patterns are assigned DFS codes that are indexed as a tree, and associated with various metadata, including identifiers for queries that use the pattern, cardinality estimations, partition identifiers, etc. Three partitioning methods are based on these patterns, with partitions stored locally in gStore. "Vertical partitioning" indexes all instances of a given pattern on the same machine. "Horizontal partitioning" distributes instances of the same pattern across various machines based on its constants. "Mixed partitioning" combines the two. Basic graph patterns are decomposed into frequent sub-patterns, where the join order and algorithms are selected to reduce communication costs.

Leon [83] (2019) is an in-memory distributed RDF store based on a master–slave architecture. Triples are partitioned based on the characteristic set of their subject; the characteristic sets are ordered in terms of the number of triples they induce, and assigned to machines with the goal of keeping a good balance. Indexes (similar to those of Stylus [96]) are

built, including a bidirectional index between subjects and their characteristic sets, an index to find characteristic sets that contain a given set of properties, and indexes to evaluate certain triple patterns. A multi-query optimization technique is implemented where, given a workload (a set) of queries, the method searches for an effective way to evaluate and share the results for common subqueries – in this case, based on characteristic sets – across queries.

SPT+VP [95] (2019) is a distributed RDF store based on the principle of combining two partitioning techniques. First, a modified property table scheme is used for storage, where one table is maintained with a column for subject and all properties in the RDF graph; instead of storing multi-valued properties in multiple rows, as in Sempala's unified property table, such values are stored as lists nested in the given row. The property table is then split (vertically) into multiple tables, similar to a clustering-based definition of a property table, but where a subject may appear in multiple tables. This "subset property table" approach is combined, secondly, with vertical partitioning storage. Given a SPARQL query, joins are reordered based on global statistics, with the property tables used for S–S joins and vertical partitioning used for other joins. The query is then translated into Spark SQL for execution.

StarMR [236] (2019) is a distributed RDF store that centers around optimizations for star joins. A graph-based storage scheme is employed, where for each node in the graph, its outward edges are represented in an adjacency list; this then supports efficient evaluation for S–S star joins. No indexing is provided, where the system targets batch-based (e.g., analytical) processing. A basic graph pattern is then decomposed into (star-shaped) sub-patterns, which are evaluated and joined. Hadoop is then used to join the results of these individual sub-patterns. Optimizations include the use of characteristic sets to help filter results, and the postponement of Cartesian products, which are used to produce the partial solutions for star joins including the non-join variables; these partial solutions are not needed if the corresponding join value is filtered elsewhere.

DISE [107] (2020) is an in-memory, distributed RDF store that conceptualizes an RDF graph as a 3-dimensional binary tensor, similar to local approaches such as BitMat; however, physical representation and storage is based on dictionary encoded triples. Partitioning is based on slicing the tensor, which is equivalent to a triple-based partitioning. Joins are evaluated starting with the triple pattern with the fewest variables. SPARQL queries are supported through the Jena (ARQ) query library and evaluated using Spark.

DP2RPQ [237] (2020) is an RDF store built on a distributed graph processing framework with support for regular path queries (RPQs), which form the core of SPARQL's property paths. Unlike the standard RPQ semantics, the evaluation returns the "provenance" of the path, defined to be the subgraph induced by matching paths. Automata are used to represent the states and the potential transitions of paths while evaluating the RPQ, and are thus used to guide a navigation-based evaluation of the RPQ implemented by passing messages between nodes in the framework. Optimizations include methods to filter nodes and edges that cannot participate in the solutions to the RPQ, compression techniques on messages, as well as techniques to combine multiple messages into one. DP2RPQ is implemented on Spark's GraphX.

Triag [162] (2020) is a distributed RDF store that optimizes for triangle-based (sub)-patterns in queries. Two types of triangular RDF subgraphs are extracted using Spark: cyclic ones (e.g., $(a, p, b), (b, q, c), (c, r, a)$) and (directed) acyclic ones (e.g., $(a, p, b), (b, q, c), (a, r, c)$). The predicates of such subgraphs are extracted, ordered, hashed, and indexed in a distributed hash table using the predicate-based hash as key and the three nodes (e.g., a, b, c) as value. An encoding is used to ensure that the ordering of predicates is canonical for the pattern (assuming that nodes are variables) and that the subgraph can be reconstructed from the node ordering. Parallel versions of hash joins and nested loop joins are supported, where triangular subqueries can be pushed to the custom index. Queries are executed over Spark. Support for inferencing is also described.

WISE [84] (2020) is a distributed RDF store using workload-aware query-based partitioning. The system follows a master–slave architecture. Queries processed by the master are also analyzed in terms of workload: common sub-patterns are extracted from a generalized version of the queries where constant subject and object nodes are first converted to variables. Query-based partitioning is applied so that common sub-patterns can be pushed to individual machines. Partitioning is dynamic, and may change as queries are received. A cost model is thus defined for the dynamic partitioning, taking into account the benefits of the change in partitioning, the cost of migrating data, and potential load imbalances caused by partition sizes; a greedy algorithm is then used to decide on which migrations to apply. The system uses Leon – an in-memory distributed RDF store discussed previously – for underlying storage and indexing.

gSmart [52] (2021) is a distributed RDF store that is capable of leveraging both GPUs and CPUs in a distributed setting. In order to take advantage of faster access for GPU memory despite its limited capacity, the LSpM storage system is used, which allows for loading compressed matrices for particular predicates and edge directions, as relevant for the query; matrices are encoded row-wise and column-wise, representing edge direction, in a compressed format, and can be partitioned for parallel computation. "Heavy queries" involving triple patterns with variable subjects and objects are then delegated to GPU computation, while "light queries" are run on CPU, where intermediate results are then combined to produce the final results on the CPU. Basic graph patterns are compiled into linear algebra operations that are efficiently computable on GPUs, with additional optimizations applied to process multi-way star joins.

A.3 Trends

We remark on some general trends based on the previous survey of local and distributed systems.

In terms of local systems, earlier approaches were based on underlying relational stores given that their implementations were already mature when interest began to coalesce around developing RDF stores. Thus, many of these earlier stores could be differentiated in terms of the relational schema (triple table, vertical partitioning, property tables, etc.) used to represent and encode RDF graphs. Systems that came later tended to rather build custom native storage solutions, optimizing for specific characteristics of RDF in terms of its graph structure, its fixed arity, etc.; relating to the fixed arity, for example, native stores began to develop complete indexes, by default, that would allow efficient lookups for any triple pattern possible. Also, many engines began to optimize for star-joins, which are often used to reconstruct n-ary relations from RDF graphs. Engines would soon start to explore graph-inspired storage and indexing techniques, including structural indexes, compressed adjacency lists, etc. A more recent trend – likely following developments in terms of hardware – has been an increased focus on in-memory stores using compact representations and compressed tensor-based representations of graphs that enable GPU-based hardware acceleration. Another recent development has been the application of worst-case optimal join algorithms for evaluating basic graph patterns, as well as techniques for translating queries into operations from linear algebra that can be efficiently evaluated on GPUs.

With respect to distributed RDF stores, in line with an increased demand for managing RDF graphs at very large scale, proposals began to emerge around 2007 regarding effective ways to store, index and query RDF over a cluster of machines.[19] Initial proposals were based on existing native stores, which were extended with triple/quad-based partitioning and distributed join processing techniques to exploit a cluster of machines. A second trend began to leverage

---

[19] We highlight that decentralized proposals for managing RDF graphs existed before this, including federated systems, P2P systems, etc., but are not considered in-scope here.

the maturation and popularity of "Big Data" platforms, including distributed processing frameworks like Hadoop and later Spark, and distributed NoSQL stores like Cassandra, HBase, MongoDB, etc., in order to build distributed RDF stores. During this time, graph-based and later query-based partitioning methods began to emerge. Like in the local case, more and more in-memory distributed RDF stores began to emerge. Another trend was to explore the use of distributed graph processing frameworks – that offer a vertex-based computation and messaging paradigm – for evaluating queries over RDF. A very recent trend is towards using both CPUs and GPUs in a distributed environment in order to enable hardware acceleration on multiple machines.

While proposed solutions have clearly been maturing down through the years, and much attention has been given to evaluating basic graph patterns over RDF, some aspects of SPARQL query processing have not gained much attention. Most stores surveyed manage triples rather than quads, meaning that named graphs are often overlooked. A key feature of SPARQL – and of graph query languages in general – is the ability to query paths of arbitrary length, where optimizing property paths in SPARQL has not received much attention, particularly in the distributed setting. Many works also focus on a WORM (write once, read many) scenario, with relatively little attention paid (with some exceptions) to managing dynamic RDF graphs.

A final aspect that is perhaps not well-understood is the trade-off that exists between different proposals, what precisely are their differences on a technical level (e.g., between relational- and graph-based conceptualizations), and which techniques perform better or worse in which types of settings. In this regard, a number of benchmarks have emerged to try to compare RDF stores in terms of performance; we will discuss these in the following section.

B SPARQL Benchmarks for RDF Stores

We now discuss a variety of SPARQL benchmarks for RDF stores. We speak specifically of SPARQL benchmarks since benchmarks for querying RDF either came after the standardization of SPARQL (and thus were formulated in terms of SPARQL), or they were later converted to SPARQL for modern use. The discussion herein follows that of Saleem et al. [200], who analyze different benchmarks from different perspectives. We first discuss the general design principles for benchmarks, and then survey specific benchmarks.

B.1 SPARQL Benchmark Design

SPARQL query benchmarks consist of three elements: RDF graphs (or datasets), SPARQL queries, and performance measures. We first discuss some design considerations regarding each of these elements.

Datasets The RDF graphs and datasets proposed for use in SPARQL benchmarks are of two types: *real-world* and *synthetic*. Both have strengths and weaknesses.

Real-world graphs reflect the types of graphs that one wishes to query in practice. Graphs such as DBpedia, Wikidata, YAGO, etc., tend to be highly complex and diverse; for example, they can contain hundreds, thousands or tens of thousands of properties and classes. Presenting query performance over real-world graphs is thus a relevant test of how a store will perform over RDF graphs found in practice. Certain benchmarks may also include a number of real-world graphs for the purposes of distributed, federated or even decentralized (web-based) querying [205].

Synthetic graphs are produced using specific generators that are typically parameterized, such that graphs can be produced at different scales, or with different graph-theoretic properties. Thus synthetic graphs can be used to test performance at scales exceeding real-world graphs, or to understand how particular graph-theoretic properties (e.g., number of properties, distributions of degrees, cyclicity, etc.) affect performance. Synthetic graphs can also be constructed to emulate certain properties of real-world graphs [65].

A number of measures have been proposed in order to understand different properties of benchmark graphs. Obvious ones include basic statistics, such as number of nodes, number of triples, number of properties and classes, node degrees, etc. [198,65]. Other (less obvious) proposals of measures include *structuredness* [65], which measures the degree to which entities of the same class tend to have similar characteristic sets; *relationship specialty* [191], which indicates the degree to which the multiplicity of individual properties varies for different nodes, etc. Observations indicate that the real-world and synthetic graphs that have been used in benchmarks tend to vary on such measures, with more uniformity seen in synthetic graphs [65,191,200]. This may affect performance in different ways; e.g., property tables will work better over graphs with higher structuredness and (arguably) lower relationship specialty.

SPARQL Queries The second key element of the benchmark is the queries proposed. There are three ways in which the queries for a benchmark may be defined:

- *Manually-generated*: The benchmark designer may manually craft queries against the RDF graph, trying to balance certain criteria such as query features, complexity, diversity, number of results, etc.
- *Induced from the graph*: The queries may be induced from the RDF graph by extracting sub-graphs (e.g., using some variation on random walks), with constants in the sub-graphs replaced by variables to generate basic graph patterns.

– *Extracted from logs*: The queries to be used may be extracted from real-world SPARQL logs reflecting realistic workloads; since logs may contain millions of queries, a selection process is often needed to identify an interesting subset of queries in the log.

Aside from concrete queries, benchmarks may also define query templates, which are queries where a subset of variables are marked as placeholders. These placeholders are replaced by constants in the data, typically so that the resulting partially-evaluated query still returns results over the RDF graph. In this way, each template may yield multiple concrete queries for use in the benchmark, thus smoothing variance for performance that may occur for individual queries.

Queries can vary in terms of the language considered (SPARQL 1.0 vs. SPARQL 1.1) and the algebraic features used (e.g., projection, filters, paths, distinct, etc.), but also in terms of various measures of the complexity and diversity of the queries – and in particular, the basic graph patterns – considered. Some basic measures to characterize the complexity and diversity of queries in a benchmark include the number of queries using different features, measures for the complexity of the graph patterns considered (e.g., number of triple patterns, number of variables, number of joins variables, number of cyclic queries, mean degree of variables, etc.), etc. Calculating such measures across the queries of the benchmark, a high-level *diversity score* can be computed for a set of queries [200], based on the average coefficient of variation (dividing the mean by the standard deviation) across the measures.

Performance Measures The third key element of a benchmark is the performance measures used. Some benchmarks may be provided without a recommended set of measures, but at the moment in which a benchmark is run, the measures to be used must be selected. Such measures can be divided into four categories [200]:

– *Query Processing Related*: The most important dimension relating to query processing relates to runtimes. A benchmark usually contains many queries, and thus reporting the runtime for each and every query is often too fine-grained. Combined results can rather be presented with measures like Query Mix per Hour (QMpH), Queries per Second (QpS), or measures over the distributions of runtimes (max, mean, percentile values, standard deviation, etc.). Other statistics like the number of intermediate results generated, disk/memory reads, resource usage, etc., can be used to understand lower-level performance issues during query processing load [206].
– *Data Storage Related:* This category includes measures like data loading time, storage space, index sizes, etc. Often there is a space–time trade-off inherent in different approaches, where more aggressive indexing can help

to improve query runtimes but at the cost of space and more expensive updates. Hence these measures help to contextualize query-processing related measures.
– *Result Related:* Some systems may produce partial results for a query based on fixed thresholds or timeouts. An important consideration for a fair comparison between two RDF engines relates to the results produced in terms of correctness and completeness. This can often be approximately captured in terms of the number of results returned, the number of queries returning empty results (due to timeouts), the recall of queries, etc.
– *Update Related:* In real-world scenarios, queries are often executed while the underlying data are being updated in parallel. While the previous categories consider a read-only scenario, benchmarks may also record measures relating to updates [68, 56]. Measures may include the number of insertions or deletions per second, the number of read/write transactions processed, etc.

Often a mix of complementary measures will be presented in order to summarize different aspects of the performance of the tested systems.

B.2 Synthetic Benchmarks

We now briefly survey the SPARQL benchmarks that have been proposed and used in the literature, and that are available for download and use. We start with benchmarks based on synthetic data.

LUBM (Lehigh) [85] (2005) creates synthetic RDF graphs that describe universities, including students, courses, professors, etc. The number of universities described by the graph is a parameter that can be changed to increase scale. The benchmark includes 14 hand-crafted queries. LUBM further includes an OWL ontology to benchmark reasoning, though often the benchmark is run without reasoning.

BSBM (Berlin) [35] (2009) is based on an e-commerce use-case describing entities in eight classes relating to products. The number of products can be varied to produce RDF graphs of different scales. A total of 12 query templates are defined with a mix of SPARQL features. The benchmark is also given in SQL format, allowing to compare RDF stores with RDBMS engines.

SP^2Bench [206] (2009) creates synthetic RDF graphs that emulate an RDF version of the DBLP bibliographic database. Various distributions and parameters from the DBLP data are extracted and defined in the generator. A total of 17 queries are then defined for the benchmark in both SPARQL and SQL formats.

BowlognaBench [62] (2012) creates synthetic RDF graphs inspired by the Bologna process of reform for European universities. The dataset describes entities such as students, professors, theses, degrees, etc. A total of 13 queries are defined that are useful to derive analytics for the reform process.

WatDiv [12] (2014) provides a data generator that produces synthetic RDF graphs with an adjustable value of structuredness, and a query template generator that generates a specified number of query templates according to specified constraints. The overall goal is to be able to generate diverse graphs and queries.

LDBC-SNB [68] (2015) is a benchmark based on synthetically generated social networking graphs. Three workloads are defined: *interactive* considers both queries and updates in parallel; *business intelligence* considers analytics that may touch a large percentage of the graph; *algorithms* considers the application of graph algorithms.

TrainBench [222] (2018) is a synthetic benchmark inspired by the use-case of validating a railway network model. The graph describes entities such as trains, switches, routes, sensors, and their relations. Six queries are defined that reflect validation constraints. TrainBench is expressed in a number of data models and query languages, including RDF/SPARQL and SQL.

B.3 Real-World Benchmarks

Next we survey benchmarks that are based on real-world datasets and/or queries from real-world logs.

DBPSB (DBpedia) [159] (2011) clusters queries from the DBpedia logs, generating 25 query templates representative of common queries found. These queries can then be evaluated over DBpedia, where a dataset of 153 million triples is used for testing, though smaller samples are also provided.

FishMark [24] (2012) is based on the FishBase dataset and is provided in RDF and SQL formats. The full RDF graph uses 1.38 billion triples, but a smaller graph of 20 million triples is used for testing. In total, 22 queries from a log of real-world (SQL) queries are converted to SPARQL.

BioBenchmark [247] (2014) is based on queries over five real-world RDF graphs relating to bioinformatics – Allie, Cell, DDBJ, PDBJ and UniProt – with the largest dataset (DDBJ) containing 8 billion triples. A total of 48 queries are defined for the five datasets based on queries generated by real-world applications.

FEASIBLE [199] (2015) generates SPARQL benchmarks from real-world query logs based on clustering and feature selection techniques. The framework is applied to DBpedia and Semantic Web Dog Food (SWDF) query logs and used to extract 15–175 benchmark queries from each log. The DBpedia and SWDF datasets used contain 232 million and 295 thousand triples, respectively.

WGPB [100] (2019) is a benchmark of basic graph patterns over Wikidata. The queries are based on 17 abstract patterns, corresponding to binary joins, paths, stars, triangles, squares, etc. The benchmark contains 850 queries, with 50 instances of each abstract pattern mined from Wikidata using guided random walks. Two Wikidata graphs are given: a smaller one with 81 million triples, and a larger one with 958 million triples.

B.4 Benchmark Comparison and Results

For a quantitative comparison of (most of) the benchmarks mentioned here, we refer to the work by Saleem et al. [200], which provides a detailed comparison of various measures for SPARQL benchmarks. For benchmarks with results comparing different RDF stores, we refer to the discussion for (italicizing non-RDF/SPARQL engines):

– BSBM [35] (2009) with results for Jena, RDF4J, Virtuoso and *MySQL*;
– SP^{2}Bench [206] (2009) including results for Kowari, Jena, RDF4J, Redland and Virtuoso.
– DBPSB [159] (2011) with results for GraphDB, Jena, RDF4J and Virtuoso;
– BowlognaBench [62] (2012) with results for 4store, dipLODocus, RDF-3X and Virtuoso.
– FishMark [24] (2012) with results for Virtuoso, *MySQL* and *Quest*;
– BioBench [247] (2014) with results for 4store, Blazegraph, GraphDB, Kowari and Virtuoso.
– WatDiv [12] (2014) with results for 4store, gStore, RDF-3X, Virtuoso and *MonetDB*;
– FEASIBLE [199] (2015) with results for GraphDB, Jena, RDF4J and Virtuoso;
– LDBC-SB [68] (2015), with results for SparkSee and Virtuoso;
– TrainBench [222] (2018) with results for Jena, RDF4J, *Neo4j* and *SQLite*, among others.

For a performance comparison of eleven distributed RDF stores (SHARD, H2RDF+, CliqueSquare, S2X, S2RDF, AdPart, TriAD, H-RDF-3x, SHAPE, gStore-D and DREAM) and two local RDF stores (gStore and RDF-3X) over various benchmarks (including LUBM and WatDiv), we refer to the experimental comparison by Abdelaziz et al. [3].