RDF Data Storage and Query Processing Schemes: A Survey

MARCIN WYLOT and MANFRED HAUSWIRTH, TU Berlin / Fraunhofer FOKUS, Germany
PHILIPPE CUDRÉ-MAUROUX, University of Fribourg, Switzerland
SHERIF SAKR, University of Tartu, Estonia King Saud bin Abdulaziz University for Health Sciences, Saudi Arabia

The Resource Description Framework (RDF) represents a main ingredient and data representation format for Linked Data and the Semantic Web. It supports a generic graph-based data model and data representation format for describing things, including their relationships with other things. As the size of RDF datasets is growing fast, RDF data management systems must be able to cope with growing amounts of data. Even though physically handling RDF data using a relational table is possible, querying a giant triple table becomes very expensive because of the multiple nested joins required for answering graph queries. In addition, the heterogeneity of RDF Data poses entirely new challenges to database systems. This article provides a comprehensive study of the state of the art in handling and querying RDF data. In particular, we focus on data storage techniques, indexing strategies, and query execution mechanisms. Moreover, we provide a classification of existing systems and approaches. We also provide an overview of the various benchmarking efforts in this context and discuss some of the open problems in this domain.

CCS Concepts: • **Information systems** → *Data management systems*; *Database design and models*; *Graph-based database models*; *Data model extensions*; *Semi-structured data*; *Query languages*;

Additional Key Words and Phrases: RDF, SPARQL, semi-structured data

ACM Reference format:
Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. 2018. RDF Data Storage and Query Processing Schemes: A Survey. *ACM Comput. Surv.* 51, 4, Article 84 (September 2018), 36 pages.
https://doi.org/10.1145/3177850

1 INTRODUCTION

The nature of the World Wide Web has evolved from a web of linked documents to a web that also includes Linked Data (Bizer et al. 2009). Traditionally, we were able to publish documents on the Web and create links between them. Those links, however, only allow the document space to be traversed without understanding the relationships between the documents and without linking to particular pieces of information. Linked Data provides the ability to create meaningful links between pieces of data on the Web (Berners-Lee et al. 2001). The adoption of Linked Data technologies has shifted the Web from a space of connecting documents to a global space where pieces of

The work of Sherif Sakr has been supported by Estonian Research Council Grant No. MOBTT75.
Authors' addresses: M. Wylot and M. Hauswirth, TU Berlin / Fraunhofer FOKUS, Germany; emails: {m.wylot, manfred.hauswirth}@tu-berlin.de; P. Cudré-Mauroux, University of Fribourg, Switzerland; email: phil@exascale.info; S. Sakr, University of Tartu, Estonia and King Saud Bin Abdulaziz University for Health Science, Saudi Arabia; email: sherif.sakr@ut.ee.

https://doi.org/10.1145/3177850

data from different domains are semantically linked and integrated to create a global Web of Data. Linked Data enables operations to deliver integrated results as new data is added to the global space. This opens up new opportunities for applications such as search engines, data browsers, and various domain-specific systems. The Web of Linked Data allows applications to operate on an unbounded and machine-processable space of semi-structured data, thus enabling them to return comprehensive results as new data appears on the Web (Bizer et al. 2009). Moreover, this allows applications to join data from multiple independent and distributed data collections.

RDF[1] represents an emerging data model that provides the means to describe resources in a semi-structured manner for these applications. In practice, RDF is gaining widespread momentum and usage in different domains, such as the Semantic Web, Linked Data, Open Data, social networks, digital libraries, bioinformatics, or business intelligence. As an example of this trend, a growing number of ontologies and knowledge bases storing millions to billions of facts, such as *DBpedia*,[2] *Probase*,[3] and *Wikidata*,[4] are now publicly available. In addition, key search engines like Google and Bing are providing better support for RDF. In principle, RDF is designed to flexibly model schema-free information that represents data objects as *triples* in the form (S, P, O), where S represents a subject, P represents a predicate, and O represents an object. A triple indicates a relationship between S and O captured by P. Consequently, a collection of triples can be modelled as a directed graph where the graph vertices denote subjects and objects while graph edges are used to denote predicates.

The wide adoption of the RDF data model has called for efficient and scalable RDF infrastructures. As a response to this call, many centralized and distributed RDF systems have been presented to tackle these challenges. This article provides a comprehensive survey of the various approaches and design strategies for implementing RDF data storage and querying systems. The remainder of this article is organized as follows. We start by providing some background information in Section 2. We introduce our taxonomy and classification of the various systems according to several design dimensions in Section 3. A detailed description of centralized RDF systems is presented in Section 4, while distributed RDF systems are covered in Section 5. We discuss some of the benchmarking efforts in this domain in Section 7 before we conclude the article and discuss a number of open challenges in Section 8.

2 BACKGROUND INFORMATION

RDF is a graph-based format that allows us to define named links between resources in the form of triples *Subject, Predicate, Object*, also called statements. A statement expresses a relationship (defined by a predicate) between resources (subject and object). The relationship is always from subject to object (it is directional). The same resource can be used in multiple triples playing the same or different roles; e.g., it can be used as a subject in one triple as well as a predicate or an object in another one. This ability enables definition of multiple connections between the triples, hence creation of a connected graph of data. Such graph can be represented as nodes that stands for the resources and edges capturing the relationships between the nodes. Figures 1 and 2 depict simple exemplary visualization of RDF graphs.

The basic triple representation of pieces of data that are combined together results in larger RDF graphs. Such large amounts of data are made available as Linked Data where datasets are interlinked and published on the Web. Elements appearing in the triples (subjects, predicates, objects) can be of one of the following types:

[1] https://www.w3.org/RDF/.
[2] http://wiki.dbpedia.org/.
[3] https://www.microsoft.com/en-us/research/project/probase/.
[4] https://www.wikidata.org/.

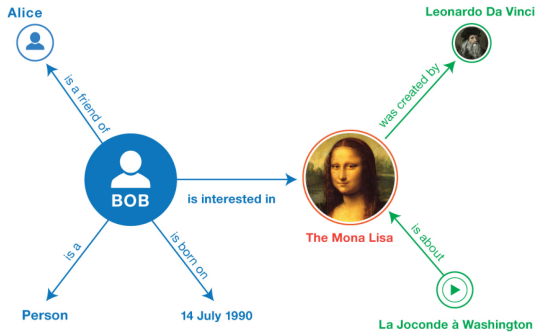

Fig. 1. An example of a graph of triples. Consortium (2014b).

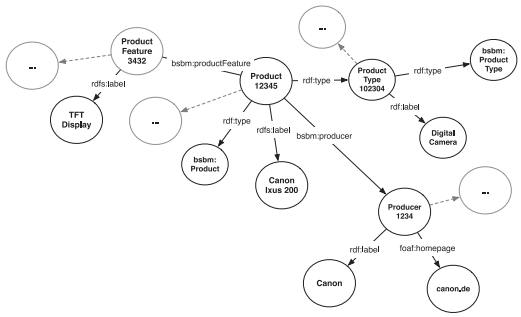

Fig. 2. An example of an RDF subgraph using the subject, predicate, and object relations given by the sample data.

- *IRI (International Resource Identifier)*: identifies a resource. It provides a global identifier for a resource without implying its location or a way to retrieve it. The identifier can be re-used by others to identify the same resource. IRIs are a generalization of URIs (Uniform Resource Identifiers) that allow non-ASCII characters to be used. IRIs can appear at all three positions in a triple (subject, predicate, or object).
- *Literal*: is a basic string value that is not an IRI. It can be associated with a datatype, thus can be parsed and correctly interpreted. It is allowed only as an object of a triple.
- *Blank node*: is used to denote a resource without assigning a global identifier with an IRI; it is a local, unique identifier used within a specific RDF dataset. It is allowed as a subject and an object in a triple.

RDF provides means to co-locate triples in a subset and to associate such subsets with an IRI (Consortium 2014a). A subset of triples constitutes an independent graph of data. In practice, it provides data managers with a mechanism to create a collection of triples. A dataset can consist of multiple named graphs and no more than one unnamed (default) graph.[5] In practice, the SPARQL[6] query language has been recommended by the W3C as the standard language for querying RDF data. A SPARQL query Q specifies a graph pattern P, which is matched against an RDF graph G. The query matching process is performed via matching the variables in P with the elements of G such that the returned graph is contained in G (pattern matching). In practice, most RDF stores can

[5]https://www.w3.org/TR/rdf11-concepts/#section-dataset.
[6]https://www.w3.org/TR/sparql11-overview/.

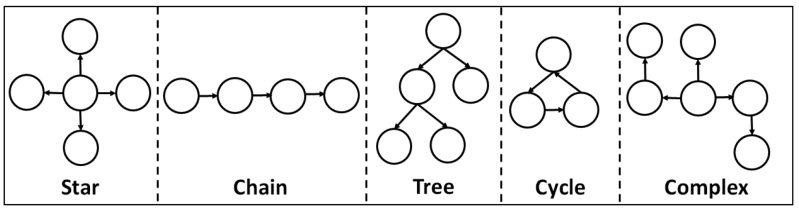

Fig. 3. Shapes of SPARQL BGP Queries.

be searched using queries that are composed of *triple patterns*. A triple pattern is much like a triple, except that S, P, and O can be replaced by variables. Similar to triples, triple patterns are modeled as directed graphs. In general, a set of triple patterns is called a *basic graph pattern (BGP)* and SPARQL queries that only include such type of patterns are called BGP queries. In practice, SPARQL BGP queries can have different shapes (Figure 3). (1) *Star* query: only consists of subject-subject joins where a join variable is the subject of all the triple patterns involved in the query. (2) *Chain* query: consists of subject-object joins where the triple patterns are consecutively connected like a chain. (3) *Tree* query: consists of subject-subject joins and subject-object joins. (4) *Cycle* query: contains subject-subject joins, subject-object joins and object-object joins. (5) *Complex* query: consist of a combination of different shapes. Therefore, answering a SPARQL BGP query is usually framed as a sub-graph pattern-matching problem (Huang et al. 2011). The main focus of this article is on the techniques and approaches that have been designed for the evaluation of conjunctive BGP queries on RDF databases. We do not focus on other RDF querying features such as the *OPTIONAL* operation (i.e., a triple pattern is matched optionally), *FILTER* expressions and string functions with regular expressions.

3 TAXONOMY AND CLASSIFICATION

The wide adoption of the RDF data model has called for efficient and scalable RDF schemes. As a response to this call, a number of systems have been designed to tackle this challenge. In general, these systems can be broadly classified into two main categories:

- *Centralized systems*: where the storage and query processing of RDF data is managed on a single node. Examples of this type of systems include (McBride 2002; Harris and Gibbins 2003; Wilkinson and Wilkinson 2006; Ma et al. 2004; Chong et al. 2005; Abadi et al. 2007; Weiss et al. 2008; Bornea et al. 2013; Neumann and Weikum 2010; Yuan et al. 2013). A main property of such systems is that they do not incur any communication overhead (i.e., they process all data locally). However, they remain limited by the computational power and memory capacities of a single machine.
- *Distributed systems*: where the storage and query processing of RDF data is managed on multiple nodes. Examples of this type of systems include (Khadilkar et al. 2012; Punnoose et al. 2015; Papailiou et al. 2012; Schätzle et al. 2013; Badam and Pai 2011; Kyrola et al. 2012; Schätzle et al. 2015; Hammoud et al. 2015). As opposed to centralized systems, distributed RDF systems are characterized by larger aggregate memory sizes and higher processing capacity. On the flip side, they might incur significant intermediate data shuffling when answering (complex) SPARQL queries, especially when queries span multiple disjoint partitions.

In general, with increasing sizes of RDF datasets, tens or hundreds of gigabytes of main memory and a high-degree of parallelism is required to rapidly satisfy the demands of *complex* SPARQL queries (i.e., queries with large numbers of triple patterns and joins)—something that is currently

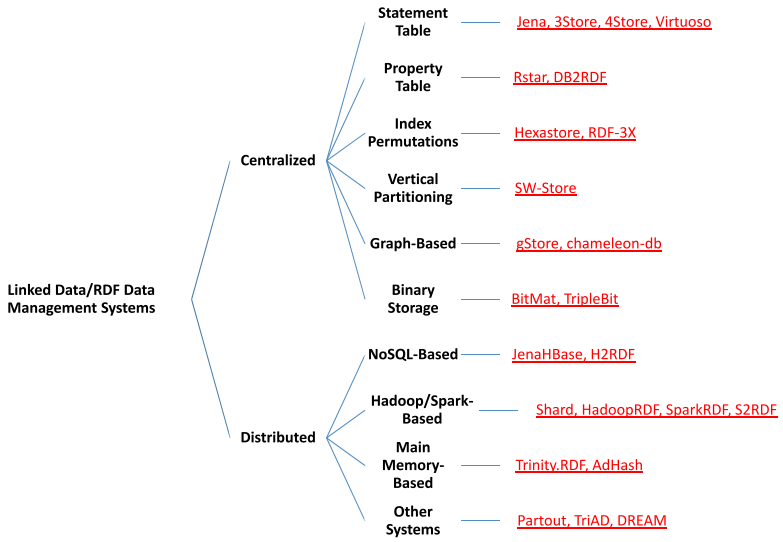

Fig. 4. Taxonomy and classification of the surveyed RDF Management Systems.

available only to high-end servers with steep prices. As a result, executing complex queries on a single node might be impractical especially when the node's main memory is dwarfed by the volume of the dataset. *Distributed* RDF systems can tackle this challenge by typically partitioning the RDF data among a set of clustered machines. In general however, in distributed systems, communication is very expensive. Therefore, intermediate data shuffling can greatly degrade query performance. Hence, reducing intermediate data shuffling is becoming one of the major challenges for distributed RDF systems.

A survey article by Özsu (2016) provided an overview of centralized RDF systems while another survey article (Kaoudi and Manolescu 2015) has mainly focused on cloud-based RDF systems. Figure 4 depicts our proposed comprehensive taxonomy as well as the classification of the various types of RDF systems. The first level of the taxonomy divides the systems based on their main design decision into two main categories: centralized and distributed. Within the centralized systems, we distinguish various storage models, i.e., the way triples are physically organized within the system (e.g., statement table, property table, graph-based store) while for the distributed systems, we classify them into categories based on the architectures or design paradigms they leverage (e.g., NoSQL, Hadoop/Spark, Main Memory). The lower level of our taxonomy list representative systems for each defined class of our taxonomy. A detailed description of the various classes of *centralized* RDF systems is presented in Section 4 while the various classes of *distributed* RDF systems are covered in Section 5.

4 CENTRALIZED RDF SYSTEMS

4.1 Statement Tables

One straightforward way to maintain RDF triples is to store triple statements in a tablelike structure. In particular, in this approach, the input RDF data is maintained as a linearized list of triples, storing them as ternary tuples. In Alexaki et al. (2001), this approach is called the "generic" approach. The RDF specification states that the objects in the graphs can be either URIs, literals, or blank nodes. Properties (predicates) are always URI references. Subject nodes can only be URIs or blank nodes. This enables to specify the underlying data types for storing subject and predicate

Subject	Predicate	Object
Product12345	rdf:type	bsbm:Product
Product12345	rdfs:label	Canon Ixus 2010
Product12345	bsbm:producer	bsbm-inst:Producer1234
...	...	...
Producer1234	rdf:label	Canon
Producer1234	foaf:homepage	http://www.canon.com
...	...	...

Fig. 5. A simple RDF storage scheme using a linearized triple representation. The illustration uses schema elements from the Berlin SPARQL Benchmark (Bizer and Schultz 2009).

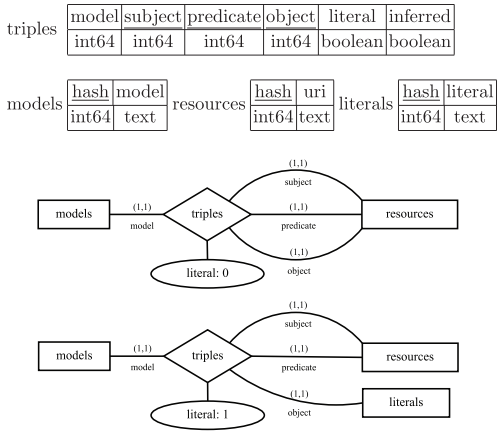

Fig. 6. Logical database design of the triple table and ER diagram showing table relationships in 3store (Harris and Gibbins 2003).

values. A common way is to store the object values using a common string representation and perform some type conversion whenever necessary. An example table showing the same data set as in Figure 2 is shown in Figure 5.

An example of *Statement Table* approach is Jena1 (McBride 2002). Jena1 uses relational databases to stores data as a statement table. URIs and Strings are encoded as IDs and two separate dictionaries are maintained for literals and resources/URIs. To distinguish literals from URI in the statement table there are two columns. In Jena2 (Wilkinson et al. 2003) the schema is denormalized and URIs and simple literals are maintained in the statement table. The dictionary tables are used only to store strings whose lengths exceed a threshold. This enables filtering operation to be performed directly on the statement table; however, it also causes higher storage consumption, since string values are stored multiple times. 3store (Harris and Gibbins 2003) is another system that stores the RDF triples into a single relational table. Such a table consists of six columns: model, which is an equivalent to RDF Graphs; hash value for subject, predicate, and object; a Boolean value indicating that the object is a literal value or an URI; and a Boolean value indicating that the triple was inferred by 3store.[7] To map the hash values to strings, i.e., URIs, models, or literals, 3store maintains three separate dictionary tables.

[7]3store supports simple inferences based on RDF entailment rules; however, the inference goes beyond the scope if this survey.

In general, the semantic information from the complete RDF graph can be exploited so that additional data can be annotated per triple and stored as a fourth element for each input triple. Harris et al. (2009) proposed another system, called 4store, where RDF triples are maintained as quads of (model, subject, predicate, object), which is highly similar to RDF Graphs/Databases. In 4store, triples assigned to the default graph are maintained in a specific model, that is, they are used in query evaluation over the default graph. 4store stores each of the quads in three indexes; in addition, it stores literal values separately. It maintains a hash table of graphs where each entry points to lists of triples in the graph. Literals are indexed through a separate hash table and they are represented as (SPO). 4store also considers two predicate-based indexes. For each predicate, two radix tries are maintained where the key is either a subject or object, and respectively object or subject and graph are stored as entries. These indices are used to filter all quads satisfying a given predicate and their subject/object. They are considered as traditional predicate indices ($P \rightarrow OS$ and $P \rightarrow SO$).

Chong et al. (2005) present an SQL-based table function *RDFMATCH*, which is designed to query the statement table of RDF data. A main advantage of this approach is that the answers of the RDFMATCH table function can be seamlessly integrated with queries on standard relational tables and also consequently processed by the various SQL's querying constructs. The main implementation of the RDFMATCH function is complied to a set of self-join operations on the underlying triple-based RDF table store. The compiled query is evaluated efficiently using B-tree indexes in addition to a set of materialized join views for specialized subject-property. Subject-Property Matrix materialized join views are utilized to reduce the query evaluation overheads that are inherent in the canonical triple-based representation of RDF. A special module is implemented to analyze the triple-based RDF table and estimate the size of the different materialized views, based on which a user can define a subset of materialized views.

Virtuoso (Erling and Mikhailov 2008) maintains data as RDF quads that consist of a graph element id, subject, predicate, and object where all the quads are maintained in a single table. Each of the attributes can be indexed in different ways. From a high-level perspective, Virtuoso is comparable to a traditional relational database with enhanced RDF support. Virtuoso adds specific types (URIs, language, and type-tagged strings) and indexes optimized for RDF. Virtuoso supports two main types of indexes. The default index corresponds to GSPO (graph, subject, predicate, object). In addition, it provides an auxiliary bitmap index (OPGS). The indexes are stored in compressed form. As strings are the most common values in the database, for example, in URIs, Virtuoso compresses these strings by eliminating common prefixes. The system does not precalculate optimization statistics. Instead, it samples data at query execution time. It also does not compute the exact statistics but just gets rough numbers of elements and estimates query cost to pick an optimal execution plan.

4.2 Index Permutations

The approach of index-permuted RDF storage exploits and optimizes traditional indexing techniques for storing RDF data. As most of the identifiers in RDF are URI strings, one optimization is to replace these strings of arbitrary lengths with unique integers. As the data is sparse and many URIs are repetitive, this technique allows us to save memory. To increase the resulting performance, the indexes are built based on shorter encoded values rather than the uncompressed values. In practice, Several systems (e.g., Weiss et al. (2008) and Neumann and Weikum (2010)) showed that it is possible to use a storage model that applies exhaustive indexing. The foundation for this approach is that any query on the stored data can be answered by a set of indices on the subject, predicates, and objects in different orders, namely, all their permutations (Figure 7), so that it allows fast access to all parts of the triples by sorted lists and fast merge-joins.

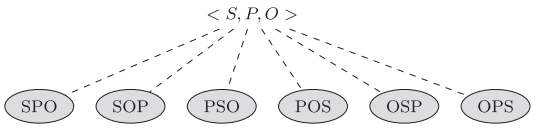

Fig. 7. Exhaustive Indexing.

Hexastore (Weiss et al. 2008) has mainly focused on generality and scalability in the design of its data storage and query processing mechanisms. Hexastore relies on maintaining the RDF data using a multiple indexing scheme (Harth and Decker 2005). It does not differentiate against any component of the RDF triples and equally treats the subject, predicate and object components. In particular, each RDF component has its special index structure. In addition, all possible combinations of the three components are indexed and materialized. Each index structure in a Hexastore is built around one RDF element (subject, predicate, or object) and determines a prioritization between the other two elements. The first level of the index is a sorted list of all subjects where each subject is associated to a list of sorted predicates. Each predicate links to a list of sorted objects. Many of the queries that may require many joins and unions in other storage systems can be answered by only using the index information. In the case where the query requests a list of subjects that are related to two particular objects through any property, the answer can be computed by merging the subject lists of the osp index. Since the subject list of this osp index is sorted, this can be done in linear time. In practice, the architectural drawback of this approach is the increase in memory consumption. Since the different combinations of possible query patterns is indexed, additional space is required due to the duplication of data. As the authors of Weiss et al. (2008) point out, less than a sixfold increase in memory consumption is required; the approach yields a worst-case fivefold increase, since for the set of spo, sop, osp, ops, pso, pos indexes, one part can always be re-used: the initial sorted list of subjects, objects, and predicates. Due to the replication of the data into the different index structures, updating and inserting into the index can become a second bottleneck.

RDF Triple eXpress (RDF-3x) (Neumann and Weikum 2010) relies on the same processing scheme of exhaustive indexing but further optimizes the data structures. In RDF-3X, the index data is stored in clustered B+ trees in lexicographic order. Moreover, indexes over count-aggregated variants for all three two-dimensional and all three one-dimensional projections are created (Neumann and Weikum 2010). The values inside the B+ tree are delta encoded (computed difference/delta between the ID attributed to the slot in the tree and the ID attributed to the previous slot) to further reduce the required amount of main memory to persist the data. Each triple (in one of the previously defined orders of spo,sop,...) is stored as a block with the maximum of 13 bytes. Since the triples are sorted lexicographically, the expected delta between two values is low, i.e., only a few bytes are consumed. Now the header of the value block contains two pieces of information: First, a flag that identifies if $value_1$ and $value_2$ are unchanged and the delta of $value_3$ is small enough to fit in the header block; second, if this flag is not set, it then identifies a case number of how many bytes are needed to decode the delta to the previous block. Figure 8 illustrates an example of the RDF-3X compression scheme where the upper part of the illustration shows the general block structure and the lower half the explicit case. Here, the flag is set to 0 meaning more than $value_3$ has changed. Case 7 identifies that for $value_1$, $value_2$, and $value_3$ exactly one byte is changed. Using this information, the deltas can be extracted and the actual value of the triple can be decoded. The query processor follows the RISC-style design philosophy (Chaudhuri and Weikum 2000) by exploiting the comprehensive set of indices on the RDF triples so that it can mostly apply merge join operations over the ordered index lists. The query optimizer uses its cost

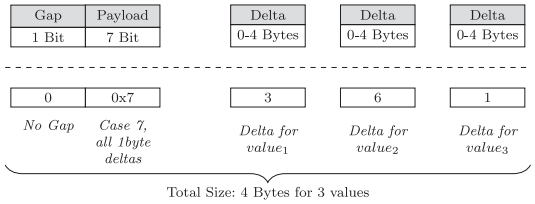

Fig. 8. RDF-3X compression example (Neumann and Weikum 2008).

model to find the most efficient query execution plan and mostly focuses reordering the join operations. In practice, selectivity estimation has a significant impact on plan generation. While this is a traditional problem in database systems, the schema-free nature of RDF data makes this issue more challenging. RDF-3X relies on a cost model to perform dynamic programming for plan enumeration. To achieve this goal, it uses two types of statistical information: (1) specialized histograms that are generic and can handle any kind of triple patterns and joins; (2) frequent join paths in the data that give more accurate estimation. During query optimization, the query optimizer uses the join-path selectivity information when available and otherwise assume independence and use the histograms information.

IBM DB2 (Bornea et al. 2013) implements only two permutations (SPO and OPS) and it stores RDF triples in four relations. The first relation (Direct Primary) stores triples indexed by subject, however this relation cannot fit multi-valued predicates. The second relation (Direct Secondary) stores data that shares the same subject and predicate. The Direct Primary relation contains a special ID for multi-valued predicates, which is used in the Direct Secondary to store values for predicates. Similarly IBM DB2 stores data indexed by objects in two relations called Reverse Primary and Reverse Secondary.

4.3 Property Tables

Since RDF does not describe any specific schema for the graph, there is no easy way to determine a set of partitioning or clustering criteria to derive a set of tables to store the information. In addition, there is no definite notion of schema stability, meaning that at any time the data schema might change, for example, when adding a new subject-object edge to the overall graph. In general, storing RDF triples in a single large statement table presents a number of disadvantages when it comes to query evaluation. In most cases, for each set of triple patterns that is evaluated in the query, a set of self-joins is necessary to evaluate the graph traversal. Since the single statement table can become very large, this can have a negative effect on query execution. Wilkinson et al. (2003) and Broekstra et al. (2002) proposed different ways to alleviate this problem by introducing the concept of property tables. In particular, instead of building one large table for all occurrences of all properties, the paper proposed two different strategies that can be distinguished into two different concepts: clustered and normalized property tables.

In principle, the main aim of clustered property tables is to group commonly accessed nodes in the graph in a single table with the goal of avoiding the high cost of many self-join operations on the large statement table encoding the RDF data. In particular, the property tables approach attempts to improve the performance of evaluating RDF queries by decreasing the cost of the join operation by reducing the number of required join operations and the size of the encoding tables of the RDF data that are involved in evaluating the RDF query. In Wilkinson and Wilkinson (2006), the use of clustered property tables is proposed for data that is stored using the Dublin Core

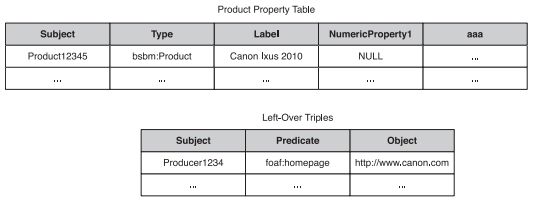

Fig. 9. Example illustrating clustered property tables. Frequently co-accessed attributes are stored together.

schema.[8] In the example shown in Figure 9, one property table for all products and a statement table for all other triples are considered. To improve the performance, a product record and all associated triples can only appear in the property table. Property tables were also implemented in Jena2 (Wilkinson et al. 2003) together with a statement table. In that context, multiple-values properties are clustered in a separate table. The system also allows to specify the type of the column in the underlying database system for the property value. This can be further leveraged for range queries and filtering. For example, the property *age* can be implemented as an integer, which can then be efficiently filtered.

DB2RDF (Bornea et al. 2013) introduced a relational encoding scheme for the RDF model that attempts to ideally encode all the predicates for each subject on a single row while efficiently maintaining the inherent variability of the different subjects. DB2RDF uses a *Direct Primary Hash* (*DPH*) wide relation where each record maintains a subject s in the entry column, with all its associated k predicates and objects stored in the $pred_i$ and val_i columns where $0 \leq i \leq k$. If subject s has more than k predicates, then a new tuple is used to store the additional attributes and the process continues untill covering and storing all the predicates for s. Since multi-valued predicates need special treatment, DB2RDF uses a second relation, *Direct Secondary Hash* (*DS*). Although the encoding scheme of DB2RDF allows one column to store multiple predicates, its hashing mechanism assure that each predicate is always assigned to the same column for the same subjects. In principle, storing all the instances of a predicate in the same column provides all the indexing advantages of traditional relational representations. In addition, storing different predicates in the same column leads to significant space savings where a relatively smaller number of database columns can be used to maintain datasets with a much bigger number of predicates (since otherwise the number of columns will be equal to the number of predicates). In principle, the *DPH* and *DS* relations primarily maintain the outgoing edges of an entity. DB2RDF also encodes the incoming edges of an entity using two additional reverse relations: the *Reverse Primary Hash* (*RPH*) and the *Reverse Secondary Hash* (*RS*). A main advantage of the DB2RDF encoding scheme is that it reduces the number of join operations for star queries (i.e., queries that ask for multiple predicates for the same subject or object).

Ontology (Fensel 2003) is a formal method for defining the types, properties, and interrelationships of the entities in a particular domain. Common components of ontology definition include *Classes* that describe collections of objects, *Attributes* that describe the properties of objects, *Individuals* that represent the instances or ground level of objects, and *Relations* that describe the ways in which classes and individuals can be related to one another. In practice, Ontology is usually presented as a taxonomy with a hierarchy of concepts where the relation between the concepts is *parent/child* or *subClass/superClass*. RStar (Ma et al. 2004) is a system that is designed to use the schema specification of the ontology definition to store ontology information and instance

[8] http://dublincore.org/.

<rdf:type>

Subject	Object
Product12345	bsbm:Product

<rdfs:label>

Subject	Object
Product12345	Canon Ixus 2010
Producer1234	Canon

<aaa>

Subject	Object
uuu	xxx
...	...

Fig. 10. Example illustrating vertical partitioning. For each existing predicate one subject-object table is created.

data in various relational tables. In particular, ontology information is encoded using tables *Class*, *SubClass*, *Property*, *SubProperty*, and *Property-Class*. In addition, a *InstanceOfClass* table is used to maintain the instances of all classes and establish the link between ontology and instance data. In RStar, each literal and each resource are assigned a unique ID and maintained in separate tables to accelerate the data retrieval and reduce the storage cost.

In general, one of the consequences of the *property tables* approach is that some information about the schema of the RDF data should be defined in advance. If the properties for a materialized type are changed during runtime, then this requires table alternations that are costly and often require explicit table-level locking. Furthermore, multi-valued attributes cannot be easily modelled using a clustered property table. If multi-valued attributes must be considered, then the defined model has to choose either to not materialize the path of the attribute or, if the sequence of the attribute is bounded, to include all possible occurrences in the materialized clustered property table.

4.4 Vertical Partitioning

SW-Store (Abadi et al. 2007) is an RDF management system that maintains RDF databases by applying a fully decomposed storage model (DSM) (Copeland and Khoshafian 1985). This approach rewrites the triple table into m tables where m is the number of unique properties in the dataset (Figure 10). Each of the m table consists of two columns. The first column stores the subjects that are described by that property, while the second column stores the object values. The subjects that are not described by a particular property are simply omitted from the table for that property. Each of the m tables is indexed by subject so that particular subjects can be retrieved quickly. In addition, fast merge join operations are exploited to reconstruct information about multiple properties for subsets of subjects. For the case of a multi-valued attribute, each distinct value is listed in a successive row in the table for that property. In practice, a main advantage of this technique is that the algorithm for creating the encoding tables is straightforward and agnostic towards the structure of the RDF dataset. SW-Store used a column-oriented database system, C-store (Stonebraker et al. 2005), to maintain the encoding tables as groups of columns instead of maintaining them as group of rows.

4.5 Graph-Based Storage

RDF naturally forms graph structures, hence one way to store and process it is through graph-driven data structures and algorithms. Therefore, some approaches have applied ideas from the graph processing world to efficiently handle RDF data. For example, gStore (Zou et al. 2014) is a graph-based RDF storage system that models RDF data as a labeled, directed multi-edge graph. In this graph, each vertex encodes a subject or an object and each triple is encoded using a directed edge from a subject to its associated object. Given a subject and an object, there may exist more than one property between them, which are represented by multiple-edges between two vertices. In gStore, the RDF graph is stored as a disk-based adjacency list table. For each class vertex in the RDF graph, gStore assigns a bitstring as its vertex signature. Therefore, the RDF graph is mapped to a data signature graph. In gStore, SPARQL queries are mapped to a subgraph matching query

over the RDF graph. During query processing, the vertices of the SPARQL query are encoded into vertex signatures and then the query is encoded into its corresponding query signature graph. Answering the SPARQL query is done by matching the vertex signature of the query graph over vertex signature of the RDF graph. In particular, gStore builds an S-tree (Deppisch 1986) for all vertices in the adjacency list tables to minimize the search space. The tree leafs correspond to vertices from the initial graph (G*), and each intermediate (parent) node is formed by performing bitwise "OR" operations on all children signatures. However, S-trees cannot support multi-way join processing; to solve this issue, the authors propose a VS-tree extension. Given an S-tree, leafs are linked according to the initial graph, and new edges are introduced depending on whether certain leafs are connected in G*. Bitwise "OR" operations over connecting edge labels of the children are performed to assign labels to such super-edges. Given a SPARQL query Q, gStore first encodes the input query and generates a query signature graph Q^*, then it finds matches of Q^* over G^* using the using the VS^*-tree by employing a top-down search strategy. Finally, gStore verifies if each match of Q^* over G^* is also a match of Q over G.

Turbo$_{HOM++}$ (Kim et al. 2015) is another graph-based approach that transforms RDF graphs into labeled graphs and applies subgraph homomorphism methods to RDF query processing. To improve its query evaluation performance, it applies type-aware transformation and tailored optimization techniques. For the type-aware transformation, it embeds the types of an entity (i.e., a subject or object) into the vertex label set so that it can eliminate corresponding query vertices/edges from the query graph. Using this appraoch, the query graph size decreases, its topology becomes simpler than the original query, and thus, this transformation improves performance accordingly by reducing the amount of graph exploration. To speed up query performance further, Turbo$_{HOM++}$ applies a series of performance optimizations as well as Non-Uniform Memory Access (NUMA)-aware parallelism for fast RDF query processing.

Attributed Multigraph-Based Engine for RDF querying (AMbER) (Ingalalli et al. 2016) is a graph-based RDF engine that represents the RDF data into multigraph where subjects/objects constitute vertices and multiple edges (predicates) can appear between the same pair of vertices. During query evaluation, SPARQL queries are also represented as multigraphs and the query answering task is transformed to the problem of subgraph homomorphism. In addition, AMbER employs an approach that exploits structural properties of the multigraph query to efficiently access RDF multigraph information and return the results.

4.6 Binary Storage

BitMat (Atre et al. 2008) is a three-dimensional (subject, predicate, object) bit matrix that is flattened in two dimensions for representing RDF triples. In this matrix, each element of the matrix is a bit encoding the absence or presence of that triple. Therefore, very large RDF triple-sets can be represented compactly in memory as BitMats. Figure 11 shows some sample RDF data and a corresponding bit matrix. The data is then compressed using D-gap compression[9] on each row level. Bitwise AND/OR operators are used to process join queries expressed as conjunctive triple patterns. In particular, BitMat creates three auxiliary tables to maintain mappings of distinct subjects, predicates, and objects to the sequence-based identifiers. Then, it groups the RDF triples by predicates and builds a subject BitMat, for each predicate group and concatenates (S,O) matrices together to get the two-dimensional BitMat. To compact the materix size, BitMat applies RLE on each subject row in the concatenated BitMat. During query processing, the BitMat representation allows fast identification of candidate result triples in addition to providing a compact representation of the intermediate results for multi-joins. The join procedure of BitMat relies on three main

[9] http://bmagic.sourceforge.net/dGap.html.

Subject	Predicate	Object
:the_matrix	:released_in	"1999"
:the_thirteenth_floor	:released_in	"1999"
:the_thirteenth_floor	:similar_plot_as	:the_matrix
:the_matrix	:is_a	:movie
:the_thirteenth_floor	:is_a	:movie

Distinct subjects: [:the_matrix, :the_thirteenth_floor]
Distinct predicates: [:released_in, :similar_plot_as, :is_a]
Distinct objects: [:the_matrix, "1999", :movie]

	:released_in			:similar_plot_as			:is_a		
:the_matrix	0	1	0	0	0	0	0	0	1
:the_thirteenth_floor	0	1	0	1	0	0	0	0	1

Note: Each bit sequence represents sequence of objects (:the_matrix, "1999", :movie)

Fig. 11. BitMat: sample bit matrix (Atre and Hendler 2009).

primitives: (1) filter, which returns BitMat by identifying a subset of triples that satisfy the triple pattern and clears bits of all other triples; (2) fold, which returns a bit-array by applying a bitwise OR on the two dimensions; (3) unfold, which returns BitMat for the bit set to 0 in the mask. In principle, the main goal of BitMat multi-join algorithm is to ensure that the intermediate result set is small using any number of join operations.

TripleBit system (Yuan et al. 2013) has been designed as a storage structure that can directly and efficiently query the compressed data. It uses a bit matrix storage structure and the encoding-based compression method for storing huge RDF graphs more efficiently. Such storage structure enables TripleBit to use merge joins extensively for join processing. In particular, TriplBit uses the Triple Matrix model where RDF triples are represented as a two dimensional bit matrix. In this matrix, each column of the matrix corresponds to an RDF triple, with only two entries of bit value associated with the subject entity and object entity of the triple. Each row is defined by a distinct entity value, with the presence in a subset of entries, representing a collection of the triples having the same entity. TripleBit sorts the columns by predicates in lexicographic order and vertically partition the matrix into multiple disjoint buckets, one per predicate. In addition, TripleBit uses two auxiliary indexing structures: (1) ID-Chunk bit matrix supports a fast search of the relevant chunks matching to a given subject or object. (2) The ID-Predicate bit matrix provides a mapping of a subject (S) or an object (O) to the list of predicates to which it relates. These indexing structures are effectively used to improve the speedup for scan and merge-join performance. TripleBit employs dynamic query plan generation algorithm to generate an optimal execution plan for a join query, aiming at minimizing the size of intermediate results as early as possible. TripleBit utilizes a unique ID for every entity to further improve the query processing efficiency as the query processor does not need to distinguish whether IDs represent subject or object entities when processing joins.

5 DISTRIBUTED RDF SYSTEMS

With increasing sizes of RDF datasets, executing complex queries on a single node has turned to be impractical in several cases specially when the node's main memory is dwarfed by the volume of the dataset. Therefore, there was a crucial need for distributed systems with a high-degree of parallelism that can satisfy the performance demands of complex SPARQL queries. As a result, several distributed RDF processing systems have been introduced where the storage and query processing of RDF data is managed on multiple nodes and by partitioning the RDF data among a set of clustered machines. In contrast to centralized systems, distributed RDF systems are characterized by larger aggregate memory sizes and higher processing capacity. However, they might incur significant intermediate data shuffling when answering complex SPARQL queries, especially when

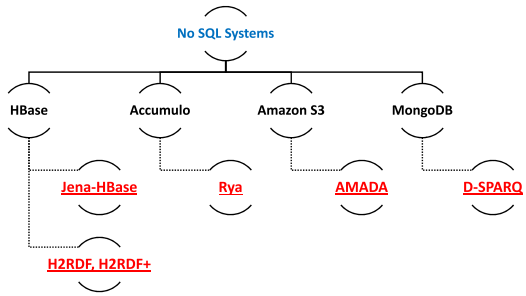

Fig. 12. NoSQL-Based RDF Systems.

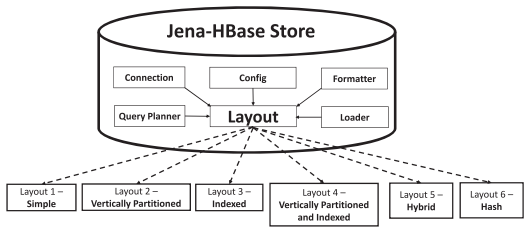

Fig. 13. The Architecture of JenaHBase System (Khadilkar et al. 2012).

queries span multiple disjoint partitions. In this section, we give an overview of various techniques and systems for efficiently querying large RDF datasets in distributed environments.

5.1 NoSQL-Based RDF Systems

The ever-growing requirement for scalability combined with new application specifications have created unprecedented challenges for traditional relational database systems. In addition to the rapid growth of information, data has become increasingly semi-structured and sparse in nature. Such emerging requirements challenged the traditional data management architectures in their need for upfront schema definition and relational-based data organization in many scenarios. In response, we have experienced the emergence of a new generation of *distributed* and scalable data storage system, referred to as NoSQL (Not Only SQL) database systems.[10] This new generation of database systems (e.g., HBase,[11] Cassandra,[12] Accumulo,[13] DynamoDB[14]) is designed with the ability to horizontally scale out over many machines, efficiently leverage the main memory and distributed indexes for data storage and to support defining new attributes or data schema dynamically (Sakr et al. 2011; Cattell 2011).

Several approaches have been exploiting the new wave of NoSQL database systems for building scalable RDF management systems. Figure 12 gives an overview of RDF systems classified according to their underlying NoSQL database design. For example, JenaHBase (Khadilkar et al. 2012) uses HBase, a NoSQL column family store, to provide various custom-built RDF data storage layouts that cover various trade-offs in terms of query performance and physical storage (Figure 13). In particular, JenaHBase designs several HBase tables with different schemas to store RDF triples.

[10] http://nosql-database.org/.
[11] https://hbase.apache.org/.
[12] http://cassandra.apache.org/.
[13] http://accumulo.apache.org/.
[14] https://aws.amazon.com/dynamodb/.

The simple layout uses three tables each indexed by subjects, predicates, and objects. For every unique predicate, the vertically partitioned layout creates two tables where each of them is indexed by subjects and objects. The indexed layout uses six tables representing the six possible combinations of indexing RDF triples, like the index permutation approaches described in Section 4.2. The hybrid layout combines both the simple and vertical partitioning (Section 4.4) layouts. The hash layout combines the hybrid layout with hash values for nodes and a separate table maintaining hash-to-node encodings. For each of these layouts, JenaHBase processes all operations (e.g., loading triples, deleting triples, querying) on a RDF graph by implicitly converting them into operations on the underlying storage layout.

H2RDF (Papailiou et al. 2012) is a distributed RDF storage system that combines a multiple-indexing scheme over HBase and the Hadoop framework. H2RDF creates three RDF indices (SPO, POS, and OSP) over the HBase store. During data loading, H2RDF collects all the statistical information that is utilized by the join planner algorithm during query processing. During query processing, the Join Planner navigates through the query graph and greedily selects the joins that need to be executed based on the selectivity information and the execution cost of all alternative join operations. H2RDF uses a join executor module that, for any join operation, chooses the most advantageous join scenario by selecting between centralized and fully distributed execution, via the Hadoop platform. Particularly, centralized joins are evaluated in a single cluster node, while distributed join operations are evaluated by launching MapReduce jobs to process them. H2RDF+ (Papailiou et al. 2013, 2014) extended the approach of H2RDF by storing all permutations of RDF indexes. Using this indexing schema, all SPARQL queries can be efficiently processed by a single index scan on the associated index. In addition, it guarantees that each join operation between triple patterns can be evaluated via merge joins that can effectively utilize the precomputed orderings.

The Rya system (Punnoose et al. 2015) has been built on top of Accumulo, a distributed key-value and column-oriented NoSQL store that supports the ordering of keys in a lexicographical ascending order. Accumulo orders and partitions all key-value pairs according to the row ID part of the key. Rows with similar IDs are grouped into the same node for efficient and faster access. Rya stores the RDF triple (subject, predicate, and object) in the Row ID part of the Accumulo tables. In addition, it indexes the triples across three separate tables (SPO, POS, and OSP) that support all the permutations of the triple pattern. These tables store the triple in the Accumulo Row ID and order the subject, predicate, and object differently for each table. This approach exploits the row-sorting mechanism of Accumulo to efficiently store and query triples across multiple Accumulo tables. SPARQL queries are evaluated using indexed nested loops join operations.

AMADA (Aranda-Andújar et al. 2012) has been presented as a platform for RDF data management that is implemented on top of the Amazon Web Services (AWS) cloud platform. It is designed as a Software-as-a-Service (SaaS), which allows users to upload, index, store, and query RDF data. In particular, RDF data is stored using Amazon Simple Storage Service (S3).[15] The S3 interface assigns a URL to each dataset, which can be later used during the query processing on EC2 nodes. To synchronize the distributed query processing AMADA uses Amazon Simple Queue Service (SQS) providing an asynchronous message-based communication. AMADA builds its own data indexes using SimpleDB, a simple database system supporting SQL-style queries based on a key-value model that supports single-relation queries, i.e., without joins. In AMADA, the query execution is performed using virtual machines within the Amazon Elastic Compute Cloud (EC2). In practice, once a query is submitted to the system, it is sent to a query processor module running on an EC2 instance, which performs a look-up to the indexes in SimpleDB to find out

[15] https://aws.amazon.com/s3.

the relevant indexes for answering the query, and evaluates the query against them. Results are written in a file stored in S3, whose URI is sent back to the user to retrieve the query answers.

CumulusRDF (Ladwig and Harth 2011) is an RDF store that provides triple pattern lookups, a linked data server and proxy capabilities, bulk loading, and querying via SPARQL. The storage back-end of CumulusRDF is Apache Cassandra, a NoSQL database management system originally developed by Facebook (Lakshman and Malik 2010). Cassandra provides decentralized data storage and failure tolerance based on replication and failover. Cassandra's data model consists of nestable distributed hash tables. Each hash in the table is the hashed key of a row and every node in a Cassandra cluster is responsible for the storage of rows in a particular range of hash keys. The data model provides two more features used by CumulusRDF: super columns, which act as a layer between row keys and column keys, and secondary indices that provide value-key mappings for columns. The index schema of CumulusRDF consists of four indices (SPO, PSO, OSP, CSPO) to support a complete index on triples and lookups on named graphs (contexts). The indices provide fast lockup for all variants of RDF triple patterns. The indices are stored in a "flat layout" utilizing the standard key-value model of Cassandra. CumulusRDF does not use dictionaries to map RDF terms but instead stores the original data as column keys and values. Thereby, each index provides a hash-based lookup of the row key, a sorted lookup on column keys and values, thus enabling prefix lookups. CumulusRDF uses the Sesame query processor[16] to provide SPARQL query functionality. A stock Sesame query processor translates SPARQL queries to index lookups on the distributed Cassandra indices; Sesame processes joins and filter operations on a dedicated query node.

D-SPARQ (Mutharaju et al. 2013) has been presented as a distributed RDF query engine on top of MongoDB, a NoSQL document database.[17] D-SPARQ constructs a graph from the input RDF triples, which is then partitioned, using hash partitioning, across the machines in the cluster. After partitioning, all the triples whose subject matches a vertex are placed in the same partition as the vertex. In other words, D-SPARQ uses hash partitioning based on subject. In addition, similar to Huang et al. (2011), a partial data replication is then applied where some of the triples are replicated across different partitions to enable the parallelization of the query execution. Grouping the triples with the same subject enables D-SPARQ to efficiently retrieve triples that satisfy subject-based star patterns in one read call for a single document. D-SPARQ also uses indexes involving subject-predicate and predicate-object. The selectivity of each triple pattern plays an important role in reducing the query runtime during query execution by reordering the individual triple patterns within a star pattern. Thus, for each predicate, D-SPARQ keeps a count of the number of triples involving that particular predicate.

5.2 Hadoop-Based RDF Systems

In 2004, Google made a seminal contribution to the Big data community by introducing the Map Reduce model (Dean and Ghemawat 2008). It is a simple and powerful programming model for developing scalable parallel applications that can process large datasets across multiple machines. In particular, MapReduce makes programmers think in a *data-centric* style allowing them to concentrate on dataset transformation while MapReduce takes care of the distributed execution and fault tolerance details, transparently (Sakr et al. 2013). The Apache Hadoop project[18] has been introduced by Yahoo! as an open source framework that implemented the concepts of MapReduce. Hadoop attracted huge interest from both the industrial and research communities. Several systems have been exploiting the Hadoop framework for building scalable RDF processing engines.

[16] http://www.openrdf.org/.
[17] https://www.mongodb.com/.
[18] http://hadoop.apache.org/.

SHARD (Rohloff and Schantz 2010) is one of the first Hadoop-based approaches that used a clause-iteration technique to evaluate SPARQL queries against RDF datasets. It is designed to exploit the MapReduce-style jobs for high parallelization of SPARQL queries. In particular, SHARD iterates over clauses in queries to incrementally bind query variables to the RDF graph nodes while conforming to all of the query constraints. In addition, an iteration algorithm is used to coordinate the iterated MapReduce jobs with one iteration for each clause in the query. In principle, the *Map* step filters each of the bindings and the *Reduce* step removes duplicates where the key value for both *Map* and *Reduce* are the bound variables in the *SELECT* clause. The initial map step identifies all feasible bindings of graph data to variables in the first query clause. The output key of the initial map step is the list of variable bindings and the output values are set to null. The intermediate MapReduce jobs continue to construct query responses by iteratively binding graph data to variables in later clauses as new variables are introduced and then joining these new bindings to the previous bound variables such that the joined bound variables align with iteratively increasing subsets of the query clauses. The intermediate steps execute MapReduce operations simultaneously over both the graph data and the previously bound variables, which were saved to disk to perform this operation. This iteration of map-reduce-join continues until all clauses are processed and variables are assigned that satisfy the query clauses. The initial reduce step removes duplicate bindings without further modifying the output of the initial map step. A final MapReduce step consists of filtering bound variable assignments to obtain just the variable bindings requested in the *SELECT* clause of the original SPARQL query.

Husain et al. (2011) describe an approach to store RDF data in Hadoop Distributed File System (HDFS) via partitioning and organizing the data files and executing dictionary encoding. In particular, this approach applies two partitioning components. The *Predicate Split (PS)* component, which splits the RDF data into predicate files. These predicate files are then fed into the *Predicate Object Split (POS)* component, which splits the predicate files into smaller files based on the type of objects. Query processing is implemented on top of Hadoop by using a heuristic cost-based algorithm that produces a query plan with the minimal number of Hadoop jobs for joining the data files and evaluating the input query. The cost of the generated query plan is bounded by the log of the total number of variables in the given SPARQL query.

The HadoopRDF system (Huang et al. 2011) has been introduced as a scale-out architecture, which combines the distributed Hadoop framework with a centralized RDF store, RDF-3X (Neumann and Weikum 2010) (see Section 4.2) for querying RDF databases. The data partitioner of HadoopRDF executes a disjoint partitioning of the input RDF graph by vertex using a graph partitioning algorithm that allows triples, which are close to each other in the RDF graph to be allocated on the same node. The main advantages of this partitioning strategy is that it effectively reduces the network communication at query time. To further reduce the communication overhead, HadoopRDF replicates some triples on multiple machines. In particular, on each worker, the data replicator decides which triples are on the boundary of its partition and replicates them based on specified n-hop guarantees. HadoopRDF automatically decomposes the input query into chunks that can be evaluated independently with zero communication across partitions and uses the Hadoop framework to combine the resulting distributed chunks. It leverages HadoopDB (Abouzeid et al. 2009) to manage the partitioning of query evaluation across high performance single-node database systems and the Hadoop framework.

The PigSPARQL system (Schätzle et al. 2013) compiles SPARQL queries into the Pig query language (Olston et al. 2008), a data analysis platform over the Hadoop framework. Pig uses a fully nested data model and provides relational style operators (e.g., filters and joins). In PigSPARQL, a SPARQL query is parsed to generate an abstract syntax tree, which is subsequently compiled into a SPARQL algebra tree. Using this tree, PigSPARQL applies various optimizations on the

algebra level such as the early evaluation of filters and using the selectivity information for reordering the triple patterns. Finally, PigSPARQL traverses the optimized algebra tree bottom up and generates an equivalent sequence of Pig Latin expressions for every SPARQL algebra operator. For query execution, Pig automatically maps the resulting Pig Latin script onto a sequence of Hadoop jobs. An advantage of taking PigSPARQL as an intermediate layer that uses Pig between SPARQL and Hadoop is being independent of the actual Hadoop version or implementation details. RAPID+ (Ravindra et al. 2011) is another Pig-based system that uses an algebraic approach for optimizing and evaluating SPARQL queries on top of the Hadoop framework. It uses a data model and algebra (the Nested TripleGroup Data Model and Algebra (NTGA) (Kim et al. 2013)), which includes support for expressing graph pattern-matching queries.

The SHAPE system (Lee and Liu 2013) uses a semantic hash partitioning approach that combines locality-optimized RDF graph partitioning with cost-aware query partitioning for processing queries over big RDF graphs. In practice, the approach utilizes access locality to partition big RDF graphs across multiple compute nodes by maximizing the intra-partition processing capability and minimizing the inter-partition communication cost. The SHAPE system is implemented on top of the Hadoop framework with the master server as the coordinator and the set of slave servers as the workers. In particular, RDF triples are fetched into the data partitioning module hosted on the master server, which partitions the data across the set of slave servers. The SHAPE system uses RDF-3X (Neumann and Weikum 2010) (see Section 4.2) on each slave server and used Hadoop to join the intermediate results generated by subqueries. For query processing, the master node serves as the interface for SPARQL queries and performs distributed query execution planning for each query received. The SHAPE system classifies the query processing into two types: intra-partition processing and inter-partition processing. The intra-partition processing is used for the queries that can be fully executed in parallel on each server by locally searching the subgraphs matching the triple patterns of the query without any inter-partition coordination. The inter-partition processing is used for the queries that cannot be executed on any partition server, of which it needs to be decomposed into a set of subqueries such that each subquery can be evaluated by intra-partition processing. In this scenario, the processing of the query would requires multiple rounds of coordination and data transfer across a set of partition servers. Thus, the main goal of the cost-aware query partitioning phase is to generate locality-optimized query execution plans that can effectively minimize the inter-partition communication cost for distributed query processing.

CliqueSquare (Goasdoué et al. 2015; Djahandideh et al. 2015) is another Hadoop-based RDF data management platform for storing and processing big RDF datasets. With the central goal of minimizing the number of MapReduce jobs and the data transfer between nodes during query evaluation, CliqueSquare exploits the built-in data replication mechanism of the Hadoop Distributed File System (HDFS). Each of its partition has three replicas by default, to partition the RDF dataset in different ways. In particular, for the first replica, CliqueSquare partitions triples based on their subject, property, and object values. For the second replica, CliqueSquare stores all subject, property, and object partitions of the same value within the same node. Finally, for the third replica, CliqueSquare groups all the subject partitions within a node by the value of the property in their triples. Similarly, it groups all object partitions based on their property values. In addition, CliqueSquare implements a special treatment for triples whose property is *rdf:type*, by translating them into an unwieldy large property partition. CliqueSquare then splits the property partition of *rdf:type* into several smaller partitions according to their object value. For SPARQL query processing, CliqueSquare relies on a clique-based algorithm, which produces query plans that minimize the number of MapReduce stages. The algorithm is based on the variable graph of a query and its decomposition into clique subgraphs. The algorithm works in an iterative way to identify cliques and to collapse them by evaluating the joins on the common variables of each

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

6 SPARK-BASED RDF SYSTEMS

The Spark project was developed as a big data processing framework that takes the performance and concepts of the Hadoop framework to the next level by loading the data in distributed main-memory and employing cheaper shuffle operations during data processing (Zaharia et al. 2010). The fundamental programming abstraction of Spark is called a *Resilient Distributed Dataset* (RDD) (Zaharia et al. 2010), which represents a logical collection of data partitioned over the nodes. In practice, maintaining RDDs as an in-memory data structures enables Spark to exploit its functional programming paradigm by enabling user's programs to load data into a cluster's memory. In addition, users are allowed to explicitly cache an RDD in memory across nodes and reuse it in multiple MapReduce-like parallel operations. Several systems have been designed to exploit the Spark framework for building scalable RDF processing engines.

S2RDF[19] (*S*PARQL on *S*park for *RDF*) (Schätzle et al. 2015) introduced a relational partitioning schema for encoding RDF data called ExtVP (the *Ext*ended *V*ertical *P*artitioning) that extends the Vertical Partitioning (VP) schema introduced by Abadi et al. (2007) (see Section 4.4) and uses a semi-join-based preprocessing to efficiently minimize query input size by taking into account the possible join correlations between the underlying encoding tables of the RDF data and join indices (Valduriez 1987). In particular, ExtVP precomputes the possible join relations between partitions (i.e., tables). The main goal of ExtVP is to reduce the unnecessary I/O operations, comparisons, and memory consumption during executing join operations by avoiding the dangling tuples in the input tables of the join operations, i.e., tuples that do not find a join partner. In particular, S2RDF determines the subsets of a *VP* table VP_{p1} that are guaranteed to find at least one match when joined with another *VP* table VP_{p2}, where $p1$ and $p2$ are query predicates. S2RDF uses this information to precompute a number of semi-join reductions (Bernstein and Chiu 1981) of VP_{p1}. The relevant semi-joins between tables in VP are determined by the possible joins that can occur when combining the results of triple patterns during query execution. Clearly, ExtVP comes at the cost of some additional storage overhead in comparison to the basic vertical partitioning techniques. Therefore, ExtVP does not use exhaustive precomputations for all the possible join operations. Instead, an optional selectivity threshold for ExtVP can be specified to materialize only the tables where reduction of the original tables is large enough. This mechanism facilitates the ability to control and reduce the size overhead while preserving most of its performance benefit. S2RDF uses the Parquet[20] columnar storage format for storing the RDF data on the Hadoop Distributed File System (HDFS). S2RDF is built on top of Spark. The query evaluation of S2RDF is based on SparkSQL (Armbrust et al. 2015), the relational interface of Spark. It parses a SPARQL query into the corresponding algebra tree and applies some basic algebraic optimizations (e.g., filter pushing) and traverses the algebraic tree bottom-up to generate the equivalent Spark SQL expressions. For the generated SQL expression, S2RDF can use the precomputed semi-join tables, if they exist, or alternatively uses the base encoding tables.

SparkRDF (Chen et al. 2014, 2015) is another Spark-based RDF engine that partitions the RDF graph into MESGs (Multi-layer Elastic SubGraphs) according to relations (R) and classes (C) by

[19] http://dbis.informatik.uni-freiburg.de/S2RDF.
[20] https://parquet.apache.org/.

building five kinds of indices (C, R, CR, RC, CRC) with different granularities to support efficient evaluation for the different query triple patterns. SparkRDF creates an index file for every index structure and stores such files directly in the HDFS, which are the only representation of triples used for the query execution, similar to the index permutation approaches described in Section 4.2. These indices are modeled as Resilient Discreted SubGraphs (RDSGs), a collection of in-memory subgraphs partitioned across nodes. SPARQL queries are evaluated over these indices using a series of basic operators (e.g., filter, join). All intermediate results are represented as RDSGs and maintained in the distributed memory to support faster join operations. SparkRDF uses a selectivity-based greedy algorithm to build a query plan with an optimal execution order of query triple patterns that aims to effectively reduce the size of the intermediate results. In addition, it uses a location-free pre-partitioning strategy that avoids the expensive shuffling cost for the distributed join operations. In particular, it ignores the partitioning information of the indices while repartitioning the data with the same join key to the same node.

The S2X (SPARQL on Spark with GraphX) (Schätzle et al. 2015) RDF engine has been implemented on top of GraphX (Gonzalez et al. 2014), an abstraction for graph-parallel computation that has been augmented to Spark (Zaharia et al. 2010). It combines graph-parallel abstractions of GraphX to implement the graph pattern-matching constructs of SPARQL. A similar approach has been followed by Goodman and Grunwald (2014) for implementing an RDF engine on top the GraphLab framework, another graph-parallel computation platform (Low et al. 2012). Naacke et al. (2016) compared five Spark-based SPARQL query processing based on different join execution models. The results showed that hybrid query plans combining partitioned join and broadcast joins improve query performance in almost all cases.

TripleRush (Stutz et al. 2015) is based on the graph processing framework Signal/Collect (Stutz et al. 2010), a parallel graph processing system written in Scala. TripleRush evaluates queries by routing partially matched copies of the query through an index graph. By routing query descriptions to data, the system eliminates joins in the traditional sense. TripleRush implements three kinds of Signal/Collect vertices: (1) *Triple Vertices* represent RDF triples; each vertex contains a subject, predicate, and Object. (2) *Index Vertices* for triple patterns that routes to triples. Each vertex contains a triple pattern (with one or more positions as wildcards); these vertices build a graph from a match-it-all triple pattern to actual tipples. (3) *Query Vertices* to coordinate the query execution process. Such vertices are created for each query executed in the system. The vertex then initiates a query traversal process through the index graph before returning the results. The most distinguished feature of TripleRush is the ability to inherently divide a query among many processing units.

6.1 Main Memory-Based Distributed Systems

Trinity.RDF (Zeng et al. 2013) has been presented as a distributed in-memory RDF system. Trinity.RDF is built on top of Trinity (Shao et al. 2013), a distributed main memory-based key-value storage system and a custom communication protocol using the Message Passing Interface (MPI) standard. In particular, Trinity.RDF provides a graph interface on top of the key-value store by randomly partitioning the RDF dataset across a cluster of machines by hashing on the graph nodes. Therefore, each machine maintains a disjoint part of the graph. For any SPARQL query, a user submits his query to a proxy. Trinity.RDF performs parallel search on each machine by decomposing the input query into a set of triple patterns and conducting a sequence of graph traversal to produce bindings for each of the triple pattern. The proxy generates a query plan and submits the plan to all the machines that maintain the RDF dataset where each machine evaluates its part of the query plan under the coordination of the proxy node. Once the bindings for all the

variables are resolved, all machines return the evaluated bindings to the proxy where the final result is computed and delivered to the end user.

In general, the way the RDF graph is partitioned can significantly impact the performance of any distributed RDF query engine. chameleon-db (Aluc et al. 2013) has been proposed as a workload-aware RDF data management system that automatically and periodically adjusts its layout of the RDF database with the aim of optimizing the query execution time and auto-tuning its performance. Such adjustment is done in a way that enables partitions to be concurrently updated without the need of stopping the query processing. Similar to gStore (Zou et al. 2014), in chameleon-db, RDF data and SPARQL queries are represented as graphs, and queries are evaluated using a subgraph matching algorithm. However, in contrast with gStore, which evaluates the queries over the entire RDF graph, chameleon-db partitions the RDF graph and prunes out the irrelevant partitions during query evaluation by using partition indexes. In practice, during the evaluation of RDF queries, it is natural that some intermediate *dormant* tuples are returned as part of the results of sub-queries; however, these tuples do not contribute to the final result, because, for instance, they may not join with intermediate results of another sub-query. The main goal of the chameleon-db partitioning strategy, called as partition-restricted evaluation (PRE), is to carefully identify the graph partitions that truly contribute to the final results to reduce the number of dormant triples that is required to be processed during query evaluation and hence improve the system performance for that workload. To prune the irrelevant partitions, chameleon-db uses an incremental indexing technique that uses a decision tree to keep track of which segments are relevant to which queries. In addition, it uses a *vertex-index*, which is a hash table that maps URIs to the subset of partitions that contain vertices of that URI and a *range-index* that keeps track of the minimum and maximum literal values within each partition for each distinct predicate.

AdHash (Harbi et al. 2015; Al-Harbi et al. 2016) is another distributed in-memory RDF engine that initially applies lightweight hash partitioning that distributes triples of the RDF triples by hashing on their subjects. AdHash attempts to improve the query execution times by increasing the number of join operations that can be executed in parallel without data communication through utilizing hash-based locality. In particular, the join patterns on subjects included in a query can be processed in parallel. The locality-aware query optimizer exploits this property to build a query evaluation plan that reduces the size of intermediate results transferred among the worker nodes. In addition, AdHash continuously monitors the data access patterns of the executed workload and dynamically adapts to the query workload by incrementally redistributing and replicating the frequently used partitions of the graphs. The main goal for the adaptive dynamic strategy of AdHash is to effectively minimize or eliminate the data communication cost for future queries. Therefore, hot patterns are redistributed and potentially replicated to allow future workloads that contain them to be evaluated in parallel by all worker nodes without any data transfer. To efficiently manage the replication process, AdHash specifies a budget constraint and uses an eviction policy for the redistributed patterns. As a result, AdHash attempts to overcome the disadvantages of static partitioning schemes and dynamically reacts with changing workloads. Figure 14 illustrates the architecture of the AdHash RDF engine. In this architecture, the master starts by partitioning the data across the worker nodes and gathering global statistical information. In addition, the master node is responsible for receiving queries from users, generating execution plans, coordinating worker nodes, collecting final results, and returning the results to users. The statistics manager maintains statistics about the RDF graph that are exploited during the global query planning and adaptive re-partitioning purposes. These statistics are distributedly gathered during the bootstrapping phase. The redistribution controller monitors the executed query workload in the form of heat maps and starts an adaptive Incremental ReDistribution (IRD) process for

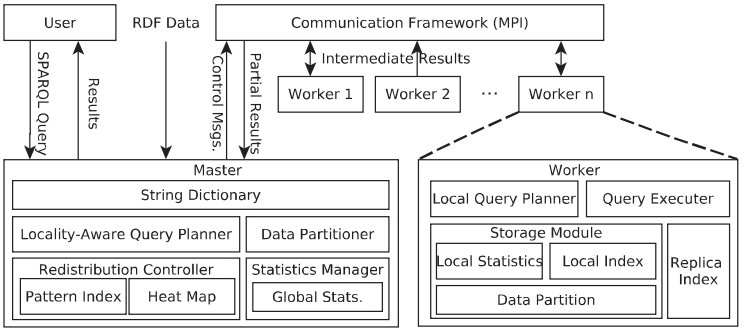

Fig. 14. The Architecture of Adhash System (Harbi et al. 2015).

hot patterns. In this process, only data that is retrieved by the hot patterns are redistributed and potentially replicated across the worker nodes (Harbi et al. 2015). In principle, a redistributed hot pattern can be answered by all workers in parallel without communication. The locality-aware query planner uses the global statistics and the pattern index from the redistribution controller to decide if a query, in whole or partially, can be processed without communication. Queries that can be fully answered without communication are planned and executed by each worker independently. However, for queries that require communication, the planner exploits the hash-based data locality and the query structure to find a plan that minimizes communication and the number of distributed joins (Harbi et al. 2015).

The Triple-Asynchronous-Distributed (TriAD) system (Gurajada et al. 2014) uses a main-memory shared-nothing architecture and is based on an asynchronous Message Passing protocol. TriAD applies a classical master-slave architecture in which the slave nodes are autonomously and asynchronously exchange messages among them to evaluate multiple join operations in parallel. Relying on asynchronous communication allows the sibling execution paths of a query plan to be processed in a freely multi-threaded fashion and only get merged (i.e., get synchronized) when the intermediate results of entire execution paths are joined. Similar to the index permutation approaches described in Section 4.2, TriAD employs six comprehensive combinations of indexing over the RDF elements. These indices are maintained into a distributed main-memory data structure where each index is first hash-partitioned according to its join key and then locally sorted in lexicographic order. Therefore, TriAD can perform efficient, distributed merge-joins over the hash-partitioned permutation lists. In addition, TriAD uses join-ahead pruning using an additional RDF summary graph, which is deployed at the master node, to prune entire partitions of triples from the SPO lists that cannot contribute to the results of a given SPARQL query. TriAD uses a bottom-up dynamic programming mechanism for join-order enumeration and considers the locality of the index structures at the slave nodes, the data exchange cost of the intermediate results, and the option to execute sibling paths of the query plan in a multi-threaded fashion, to estimate the query execution plan with the cheapest cost.

6.2 Other Distributed Systems

As Virtuoso (Erling and Mikhailov 2008) stores data in a very similar fashion as that of relational database systems (see Section 4), the clustered version of this system, namely, VirtuosoCluster also follows the strategies known from the relational databases world. The partitioning is defined at the index level, so keys from the same table can be distributed among many machines. Virtuoso implements hash partitioning with the number of logical partitions that are n times greater than

the number of physical machines. The logical partitions are assigned to the physical machines. Partitions can be re-located between machines, during this process the data is still served by the original server, but all the update operations are logged and after the re-location they are applied on the new host for the logical partition. Since the physical machines host multiple logical partitions, the allocation can be done unevenly with respect to the capacity of the physical host.

Partout (Galárraga et al. 2014) is a distributed engine that relies on a workload-aware partitioning strategy for RDF data by allowing queries to be executed over a minimum number of machines. Partout exploits a representative query load to collect information about frequently co-ocurring subqueries and for achieving optimized data partitioning and allocation of the data to multiple nodes. The architecture of Partout consists of a coordinator node and a cluster of n hosts that store the actual data. The coordinator node is responsible for distributing the RDF data among the host nodes, designing an efficient distributed query plan for a SPARQL query, and initiating query evaluation. The coordinator does not have direct access to the actual data but instead utilizes global statistics of the RDF data, generated at partitioning time, for query planning. Each of the host nodes runs a triple store, RDF-3X (Neumann and Weikum 2010) (see Section 4.2). Queries are issued at the coordinator, which is responsible for generating a suitable query plan for distributed query execution. The data is located at the hosts that are hosting the data partitions. Each host executes part of the query over its local data and sends the results to the coordinator, which will finally hold the query result. Partout's global query optimization algorithm avoids the need for a two-step approach by starting with a plan optimized with respect to the selectivities of the query predicates and then applying heuristics to obtain an efficient plan for the distributed setup. Each host relies on the RDF-3X optimizer for optimizing its local query plan.

The *D*istributed *R*DF *E*ngine with *A*daptive Query Planner and *M*inimal Communication (DREAM) system (Hammoud et al. 2015; Hasan et al. 2016) has been designed with the aim of avoiding partitioning RDF graphs and partitions SPARQL queries only, thus attempting to combine the advantages of the centralized and distributed RDF systems. DREAM stores a complete dataset at each cluster machine and employs a query planner that effectively partitions any SPARQL query, Q. In particular, DREAM partitions SPARQL queries rather than partitioning RDF datasets. This is achieved by using rule- and cost-based query planner that uses statistical information of the RDF database. Specifically, the query planner transforms Q into a graph, G, decomposes G into sets of sub-graphs, each with a basic two-level tree structure, and maps each set to a separate machine. Afterwards, all machines process their sets of sub-graphs in parallel and coordinate with each other to return the final result. No intermediate data is shuffled whatsoever and only minimal control messages and meta-data[21] are exchanged. To decide upon the number of sets (which dictates the number of machines) and their constituent sub-graphs (i.e., G's *graph plan*), the query planner enumerates various possibilities and selects a plan that will expectedly result in the lowest network and disk costs for G. This is achieved through utilizing a cost model that relies on RDF graph statistics. Using the above approach, DREAM is able to select different numbers of machines for different query types, hence, rendering it adaptive.

Cheng and Kotoulas (2015) presented a hybrid method for processing RDF that combines similar-size and graph-based partitioning strategies. With similar-size partitioning, it places similar volumes of raw triples on each computation node without a global index. In addition, graph partitioning algorithms are used to partition RDF data in a manner such that triples close to each other can be assigned to the same computation node. In practice, the main advantage of similar-size partitioning is that it allows for fast loading data while graph-based partitioning allows to achieve

[21]DREAM uses RDF-3X (Neumann and Weikum 2010) at each slave machine and communicates only triple ids (i.e., meta-data) across machines. Locating triples using triple ids in RDF-3X is a straightforward process.

efficient query processing. A two-tier index architecture is adopted. The first tier is a lightweight *primary index* that is used to maintain low loading times. The second tier is a series of dynamic, multi-level secondary indexes, evaluated during query execution, which is utilized for decreasing or removing the inter-machine data transfer for subsequent operations that maintain similar graph patterns. Additionally, this approach relies on a set of parallel mechanisms that combine the loading speed of similar-size partitioning with the execution speed of graph-based partitioning. For example, it uses fixed-length integer encoding for RDF terms and indexes that are based on hash-tables to increase access speed. The indexing process does not use network communication to increase the loading speed. The local lightweight primary index is used to support very fast retrieval and avoid costly scans while the secondary indexes are used to support non-trivial access patterns that are built dynamically, as a byproduct of query execution, to amortize costs for common access patterns.

The DiploCloud system (Wylot et al. 2011; Wylot and Cudré-Mauroux 2016) has been designed to use a hybrid storage structure by co-locating semantically related data to minimize inter-node operations. The co-located data patterns are mined from both instance and schema levels. DiploCloud uses three main data structures: molecule clusters, template lists, and a molecule index. *Molecule clusters* extend property tables to form RDF subgraphs that group sets of related URIs in nested hash-tables and to co-locate data corresponding to a given resource. *Template lists* are used to store literals in lists, like in a columnar database system. Template lists allow to process long lists of literals efficiently; therefore, they are employed mainly for analytics and aggregate queries. The molecule index serves to index URIs based on the molecule cluster to which they belong. In the architecture of DiploCloud, the Master node is composed of three main subcomponents: a key index encoding URIs into IDs, a partition manager, and a distributed query executor. The Worker nodes of the system hold the partitioned data and its corresponding local indices. The workers store three main data structures: a type index (grouping keys based on their types), local molecule clusters, and a molecule index. The worker nodes run subqueries and send results to the Master node. The data partitioner of DiploCloud relies on three molecule-based data partitioning techniques: (1) *Scope-k Molecules* manually defines the size for all molecules, (2) *Manual Partitioning* where the system takes an input manually defined shapes of molecules, (3) *Adaptive Partitioning* starts with a default shape of molecules and adaptively increases or decreases the size of molecules based on the workload. Queries that are composed of one Basic Graph Pattern (e.g., starlike queries) are executed in parallel without any central coordination. For queries requiring distributed joins, DiploCloud picks one of two executions strategies: (1) if the intermediate result set is small, then DiploCloud ships everything to the Master node that performs the join; (2) if the intermediate result set is large, then DiploCloud performs a distributed hash-join.

The EAGRE (Zhang et al. 2013) system has been presented as an *E*ntity *A*ware *G*raph comp*RE*ssion technique to encode RDF datasets using key-value storage structures that preserves the structure and semantic information of RDF graphs. The main idea of this techniques is to extract entities and entity classes from the original RDF to build a compressed RDF entity graph. EAGRE adopts a graph partition mechanism that distributes RDF data across the worker nodes and implements an in-memory index structure to efficiently accelerate the evaluation of range and order sensitive queries. In particular, the entity classes are partitioned on the computing nodes in a way that they preserve the structure locality of the original RDF graph. The evaluation process of a SPARQL query starts by identifying the entity classes using an in-memory index for the compressed RDF entity graph on a query engine. Then, the query is submitted to the worker nodes, which maintain the RDF data where the query coordinator on each worker participates in a voting process that decides the scheduling function of distributed I/O operations. EAGRE uses a distributed I/O scheduling mechanism to reduce the cost of the disk scans and the total time for the query evaluation

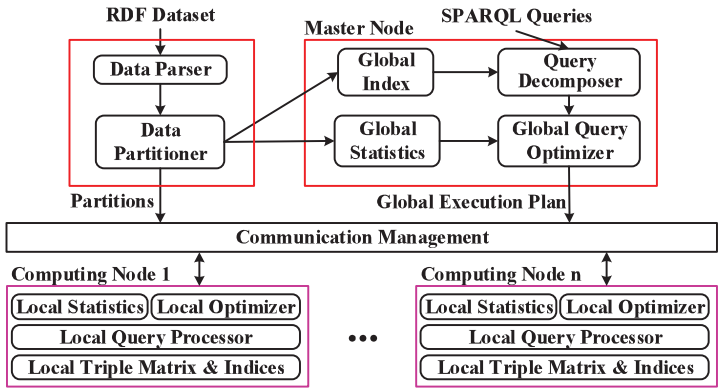

Fig. 15. The Architecture of Semstore System (Wu et al. 2014).

process. In practice, whenever some workers complete their local I/O operation, they exploit the scheduler to feed other workers with the gathered statistical information of the processed data.

Figure 15 illustrates the the architecture of the Semstore system (Wu et al. 2014) for distributed processing of RDF queries. Semstore consists of the following main components: a data partitioner, a master node, and a number of computing nodes. Semstore partitioner adopts a partitioning mechanism, Rooted Sub-Graph (RSG), which is designed to effectively localize all the queries in the shapes of a star, a chain, a tree, or a cycle that are all frequent for SPARQL queries. After partitioning the RDF graph, the data partitioner assigns each partition to one of the underlying computing nodes. The Semstore partitioner uses a k-means partitioning algorithm for assigning the highly correlated RSGs into the same node. Each computing node builds local data indices and statistics for its assigned subgraph and utilizes this information during local join processing and optimization. In addition, the data partitioner builds a global bitmap index over the vertices of the RDF graph and collects the global statistics. In Semstore, each computing node uses a centralized RDF processor, *TripleBit* (Yuan et al. 2013) (see Section 4.6), for local query evaluation. The master node is the Semstore component that receives the user query, builds the distributed query plan and coordinates distributed data transfer between the computing nodes.

Blazegraph[22] is an open-source triplestore written in Java, which is designed to scale horizontally by distributing data with dynamic key-range partitions. It also supports transactions with multiversion concurrency control relying on timestamps to detect conflicts. It maintains three RDF indices (SPA, POS, OSP) and leverages a B+Tree implementation. Those indices are dynamically partitioned into key-range shards that can be distributed between nodes in a cluster. Its scale-out architecture is based on multiple services. The shard locator service maps each key-range partition to a metadata record that allows to locate the partition. A transaction service coordinates locks to provide isolation. A client service, finally, allows to execute distributed tasks. The query execution process starts with the translation of a SPARQL query to an Abstract Syntax Tree (AST). Then, the tree is rewritten to optimize the execution. Finally, it is translated to a physical query plan, vectorized, and submitted for execution.

HDT[23] (Martínez-Prieto et al. 2012) (Header, Dictionary, Triples) has been presented as a data structure and binary serialization format. However, the framework includes querying tools and enables distributed query processing in conjunction with Hadoop (Giménez-García et al. 2015). The

[22] https://www.blazegraph.com.
[23] http://www.rdfhdt.org/.

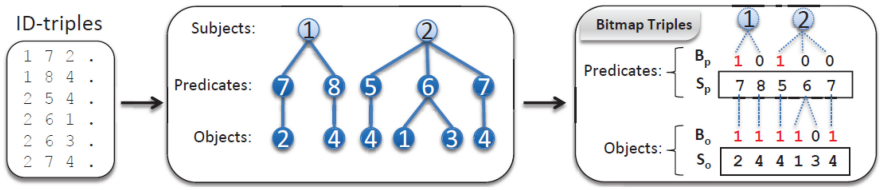

Fig. 16. HDT triples encoding (Martínez-Prieto et al. 2012).

data format consists of three components: (1) a *Header*, which holds metadata about the dataset; (2) a *Dictionary*, which encodes strings into integers that are used internally to describe triples; (3) *Triples*, which are encoded in a binary format that transform the RDF representation into multiple trees (one tree for one distinct subject) (Figure 16). Each tree consists of three levels (root/subject, list of predicates, and objects). During query execution, the patterns starting with a specified subject can be directly retrieved from the binary tree corresponding to the subject. To facilitate the evaluation of triple patterns that specify a predicate or an object, HDT introduces additional indices. SPARQL queries containing multiple triple patterns are resolved by using merge and index joins.

Peng et al. (2016) proposed a method to distribute and allocate the RDF partitions by exploring the intrinsic similarities among the structures of queries in the executed workload to reduce the number of crossing matches and the communication cost during query processing. In particular, the proposed approach mines and selects some of the frequent access patterns that reflect the characteristics of the workload. Based on the selected frequent access patterns, two fragmentation strategies, vertical and horizontal fragmentation, are used to divide RDF graphs while meeting different kinds of query processing objectives. On the one hand, the design goal of the vertical fragmentation strategy is to achieve better throughput by grouping the homomorphic matches to the same frequent access pattern into the same fragment. This strategy helps to easily filter our the irrelevant fragments during the query evaluation so that only nodes that stored relevant fragments need to be accessed to find matches while nodes that do not store relevant fragments can be used to evaluate other queries in parallel, which improves the total throughput of the system. In principle, the main focus of the vertical fragmentation strategy is to utilize the locality of SPARQL queries to improve both throughput and query response time. On the other hand, the design goal of the horizontal fragmentation strategy is to achieve better performance by putting matches of one frequent access pattern into the different fragments and distribute them among different nodes. As a result, the evaluation of one query may involve many fragments and each fragment has a few matches. In practice, the size of a fragment is often much smaller than the size of the whole data, thus finding matches of a query over a fragment explores a smaller search space than finding matches over the whole data. Therefore, with horizontal fragmentation, each node finds a few matches over some fragments, which supports the utilization of the parallelism over the clusters of nodes and reduces the query response time. For fragment allocation, a fragment affinity metric is used to measure the togetherness between the fragments and identify those that are closely related. In particular, if the affinity metric of two fragments is large, it means that these two fragments are often involved by the same query and they should be placed together to reduce the number of cross-sites joins.

6.3 Federated RDF Query Processing

The proliferation of RDF datasets created a significant need for answering RDF queries over multiple SPARQL endpoints. Such queries are referred to as *RDF federated queries*. In practice, answering

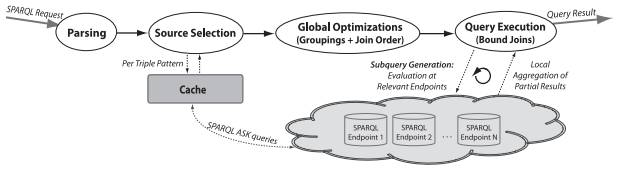

Fig. 17. FedX Query Processing Model (Schwarte et al. 2011).

such type of queries requires performing on-the-fly data integration in addition to complex graph operation over heterogeneous distributed RDF datasets. Saleem et al. (2016) perform an extensive fine-grained evaluation of federated RDF query engines. The authors show that factors like the number of selected sources, total number of used SPARQL ASK requests, and source selection time have significant impact on the query execution time. The authors conclude that the method used to select relevant sources (index-based or index-free) greatly affects the query results. Moreover, using SPARQL ASK to select sources without any caching is very expensive (two orders of magnitude difference in query execution time). To minimize the number of sub-queries most of the systems group the triple patterns that can be entirely executed on one endpoint. Several systems and approaches have been presented to deal with the challenges of federated RDF query processing. The aim of this section to highlight some of the important systems in this domain. For more information and comprehensive comparisons, we refer the reader to the surveys strictly focused on the federated RDF queries (Haase et al. 2010; Rakhmawati et al. 2013; Haase et al. 2014; Oguz et al. 2015; Saleem et al. 2016)

Figure 17 illustrates the query processing model of the FedX federated RDF query engine (Schwarte et al. 2011). In this approach, the query processing starts with the query parsing, then the system selects relevant sources for every triple pattern. Next, the system applies query optimization techniques, i.e., join ordering and grouping of triple patterns. The groups of triple patterns are then executed on the relevant endpoints. The partial results are joined with the modified nested loop join strategy to further minimize the network traffic. To select a relevant source for a triple pattern, FedX sends a SPARQL ASK query to all known endpoints. The results of such ASK queries are cached for later use. The join order optimization is based on the variable counting technique (Stocker et al. 2008). This technique estimates the cost of execution by counting free variables that are not bound through previous joins. To further minimize the cost of joins executed locally, FedX groups triple patterns that have the same set of sources on which they can be executed. They groups also must have only this source (exclusive groups). This allows them to be sent to the endpoints as a conjunctive query and minimize the cost of local joins as well as the network traffic. A set of triple patterns can be also grouped together with SPARQL UNION and sent to a remote data source instead of sending each triple pattern separately. This technique requires to keep track of the original mappings and local post-processing to return the final results. The advantage of this method is that the number of remote requests can be reduced by the factor determined by the number of grouped triple patterns. FedX is implemented in Java on top of Sesame. It is built as a Storage Layer Interface (SAIL). Nikolov et al. (2013) built a query engine on top of FedX, which is optimized for full text search and top-k queries.

SPLENDID (Görlitz and Staab 2011) uses statistics that are obtained from Vocabulary of Interlinked Datasets (VOID) (Alexander and Hausenblas 2009) descriptions to optimize the execution of federated queries. The main components of this system are: the Index Manager and the Query Optimizer (Figure 18). The Index Manager maintains the local copy of collected and aggregated statistics from remote SPARQL endpoints. The statistics for each endpoint provide information like triple count, number of distinct predicates, subjects, and objects. Moreover, SPLENDID keeps

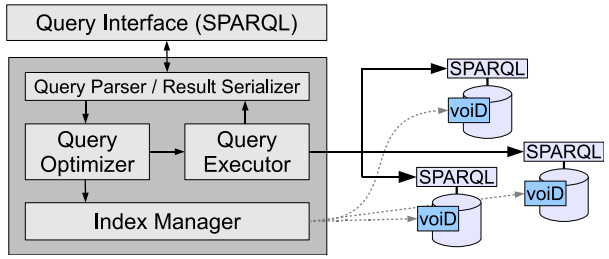

Fig. 18. The Architecture of SPLENDID (Görlitz and Staab 2011).

inverted indexes for each predicate and type. These indexes map predicates and types to a data source where it can be found and its number of occurrence within this data source. The Query Optimizer transforms the query into a syntax tree, selects a data source to federate the execution, and optimizes the order of joins. To select a data source for a triple pattern, SPLENDID uses two inverted indexes for bound predicates and types, with priority for types. In case a triple pattern has neither bound predicate nor type then all the available data sources are pre-selected. To further refine the data source selection the system sends SPARQL ASK queries with the triple pattern to the pre-selected data sources. In case one source is exclusively selected for a group of triple patterns, SPLENDID, similarly to FedX, groups them and sends to this source as a single sub-query. To join the sub-results SPLENDID implements two strategies: (1) for small result sets, the tuples are requested in parallel and a hash join is performed locally, (2) for large result sets and high selectivity of a join variable, one sub-query is executed and the join variable in the second one is repeatedly replaced with the results of the first one (bind join (Haas et al. 1997)).

The ANAPSID system (Acosta et al. 2011) leverages the concept of a non-blocking join operator Xjoin (Urhan and Franklin 2000), which is optimized to produce results quickly even in the presence of slow endpoints, and the Symmetric Hash Join (Deshpande et al. 2007). ANAPSID has four main components:

- —Catalog to store list of available endpoints along with the ontology they use and execution timeouts indicating capabilities of the endpoint.
- —Query Decomposer decomposes queries into sub-queries and chooses endpoints to execute each of them.
- —Query Optimizer to determine the exact execution plan of the sub-queries based on the collected statistics.
- —Adaptive Query Engine gathers partial results and perform final joins. These module produce the final results incrementally as the data arrives from remote endpoints. It can detect if an endpoint is blocked and modify the execution plan to prioritize the sub-queries executed on available endpoints.

To join the results from the remote endpoints ANAPSID implements the Adaptive Group Join operator *agjoin*, which is based on Xjoin and Symmetric Hash Join. These operators maintain a separate hash tables for results of each sub-query, for each endpoint. Sub-queries are executed in parallel on each endpoint, when a tuple arrives from one endpoint it is inserted to the corresponding hash table. Such hash table indexes tuples by instantiations of join variables. Then the operator immediately check other hash tables if a join operation is possible. The join is performed and the final result is produced as soon as possible, without waiting for all sub-queries to complete. ANAPSID implements a three-stage policy to manage flushing the hash table items to a secondary memory. At the first stage, it operates in the main memory, and when the main memory is full

the system moves the least recently used items to the secondary memory. When both sources are blocked, i.e., there is no item in the main memory to perform a join, then the operator start checking items in the secondary memory. Finally, the third stage is fired when all data from all sources has arrived. In this case all remaining data in the main and the secondary memory is used to produce the remaining results.

The Large-scale High-speed Distributed engine (LHD) system (Wang et al. 2013), built on top of Jena, leverages the concept of Xjoin (Urhan and Franklin 2000). Moreover, since their focus is on a highly parallel infrastructure they also leverage Double-Pipelined Hash Join (Raschid and Su 1986). They maintain multiple hash tables to join many sub-queries simultaneously. As results for sub-queries are arriving they are stored in parallel in such hash tables and at the same time probed against other hash tables to check if a join can be produced. To minimize the amount of data transferred via the network LHD used VoID statistics to estimate the cardinality of each sub-query and then, based on the cardinality, it estimates the cost. The cardinality estimation bases on the total number of triples and the total number of distinct subject and objects in a data source. Moreover, the system takes into account the number of distinct subject and objects specified in the query predicates. To generate the query plan the system chooses the cheapest possible join order the triple patterns. Then, it executes triple patterns that have specified subject or object. Finally, LHD uses dynamic programming to find optimal plan and join order to execute triple patterns with unbound subject and object.

7 BENCHMARKING RDF SYSTEMS

The Semantic Web and Linked Data community has developed several frameworks to evaluate the performance and scalability of RDF Systems. The *Lehigh University Benchmark* (LUBM)[24] (Guo et al. 2005) is one of the oldest and most popular benchmarks for Semantic Web data. It provides an ontology describing universities together with a data generator and 14 queries. Its test data consists of synthetically-generated instance data over that ontology; the data generation process is repeatable and can be scaled to an arbitrary size. The benchmark offers fourteen test queries over the data. The queries vary in terms of the input size (number of classes involved), selectivity, complexity (number of joins, matched graph degree), and inference (hierarchical and logical). The *BowlognaBench* benchmark (Demartini et al. 2012) has similar characteristics as LUBM. It provides an ontology describing the academic realm that strictly follows the setting as prescribed by the Bologna process,[25] thus the generated dataset is more realistic. The benchmark proposes 13 queries classified in eight groups that model real-world interests. The queries also include the time dimension and analytic computations.

The *Berlin SPARQL Benchmark* (*BSBM*)[26] (Bizer and Schultz 2009) is designed from an e-commerce scenario in which a set of products is provided by various vendors and consumers are posting reviews about products. The framework allows three data representations following similar semantics: RDF triples, Named Graphs data models, and relational data. The benchmark provides three use cases (query mixes): (1) *EXPLORE*: consists of search and navigation patterns of a consumer searching for a given product. (2) *EXPLORE&UPDATE*: includes the previous explore mix and adds updates on the dataset (adding new product information, reviews, offers, and deleting outdated offers). (3) *BusinessIntelligence*: consists of set of eight analytical queries.

The *SP2B* benchmark (Schmidt et al. 2009)[27] is based on a DBLP scenario. The DBLP database contains bibliographic data for Computer Science. The data is generated in an incremental and

[24] http://swat.cse.lehigh.edu/projects/lubm/.
[25] http://ec.europa.eu/education/policy/higher-education/bologna-process_en.
[26] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/.
[27] http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B.

deterministic way. The workload queries vary along different dimensions, including: query size, selectivity, output size, and different types of joins. The queries includes various graph patterns to match, as well as, clauses like OPTIONAL, UNION, DISTINCT, FILTER. The benchmark contains eleven SELECT queries and three ASK queries.

The *DBpedia SPARQL Benchmark* (*DBPSB*) (Morsey et al. 2011) generates data based on a sample from the original DBPedia. The generator duplicates all triples and changes their namespaces, hence creating a copy of the original knowledge base. The workload queries are based on queries retrieved from logs that were originally issued against DBpedia. Specifically, the authors created query templates with variables that are replaced with generated data, which results in actual queries used for benchmarking. The query templates include various characteristics including: different numbers of triple patterns, various JOIN patterns, graph pattern constructors (UNION, OPTIONAL), modifiers (e.g., DISTINCT) and filtering conditions and operators (e.g., FILTER, LANG, REGEX).

The Semantic Publishing Benchmark v2.0 (SPB)[28] is built around a media publishing scenario and is inspired by the BBC's Dynamic Semantic Publishing scenario. The benchmark uses synthetic data containing a number of annotations of media assets. The annotations consist of various properties, e.g., description, date of creation, tagged entities, and so on. The benchmark considers three kinds of dimensions when producing synthetic data: (i) clustering effect created by annotations about a single entity for a period of time, (ii) correlations created by annotations about multiple entities from reference data for a period of time, and (iii) random tagging achieved by introducing a random noise in the generated data. The query workload simulates two type of activities: *editorial* (executing insert/update/delete operations) and *aggregation* (executing select/construct/describe operations). The benchmark also allows to run multiple agents in parallel to simulate a real multi-user scenario.

The *Social Network Intelligence BenchMark*[29] (Pham et al. 2012) simulates a social network modeling relations between users and social activities (posts, comments, groups, etc.). The benchmark allows to scale the data size with the number of users in the network. The synthetic data is linked with DBPedia, e.g., by linking to real world resources. The workload queries use some of the advanced features of SPARQL 1.1 (e.g., path expressions). The queries are clustered in three workload scenarios: interactive (20 queries), update (8 update actions), and analysis (11 reports).

The WatDiv[30] (Aluç et al. 2014a) data generator allows users to control various characteristics of the generated data such as included entities, how structured is the collection, the way entities are associated and the association probability and cardinality for various types of entities. This allows to achieve different heterogeneity and structuredness of a data collection. The benchmark comes with the following workload scenarios:

- —Basic Testing: 20 queries of various complexity,
- —Extensions to Basic Testing with two use cases,
  - —Incremental Linear Testing tests the performance for queries with increasing number of triple patterns,
  - —Mixed Linear Testing tests the performance for queries of different (not necessarily increasing) number of triple patterns. The number of triple patterns ranges between 5 and 10,
- —Stress Testing offers a thorough stress test of systems.

[28] http://ldbcouncil.org/developer/spb.
[29] http://ldbcouncil.org/developer/snb.
[30] http://dsg.uwaterloo.ca/watdiv/.

Many RDF management systems present their own evaluation and comparison with related systems; however, such evaluations are inherently biased and difficult to generalize. Several benchmarking studies (Liu and Hu 2005; Schmidt et al. 2008; Sidirourgos et al. 2008; Cudré-Mauroux et al. 2013) have been conducted to provide an evaluation of a subset of the existing RDF data management systems. The results are typically available on websites of benchmarks. The biggest data collection was used in the report published within the BSBM benchmark (10M-150B triples).[31] The largest tests on non-Hadoop systems were performed for DiploCoud (Wylot and Cudré-Mauroux 2016) (up to 128 Amazon EC2 units). Sidirourgos et al. (2008) presented an independent reevaluation of the approach proposed by Abadi et al. (2007). They reported that the performance of binary tables is not always outperforming clustered property tables as the performance mainly depends on the characteristics of the RDF graph. They also reported that the gain in performance in column-store databases over row-store databases depends on the number of predicates in a data set. Cudré-Mauroux et al. (2013) presented an evaluation of different NoSQL stores (e.g., HBase, Couchbase, Cassandra) for RDF processing deployed on the Amazon EC2 infrastructure (1–16 units). The authors used four datasets of different sizes (up to one billion triples) based on the DBPedia Benchmark and BSBM. The results showed that NoSQL systems can execute simple RDF queries very efficiently.

8 CONCLUSIONS

The RDF is increasingly being adopted for modeling data in various application domains and has become a cornerstone for publishing, exchanging, sharing, and interrelating data on the Web through the Linked Data movement. In general, efficiently maintaining large volumes of RDF data is a challenging task due to the inherent heterogeneity of its structure. While RDF management systems started by borrowing form relational and centralized architectures, new physical designs have become a crucial requirement following the massive increase of available RDF data. In this article, we provided a comprehensive survey of RDF data management systems by focusing on the different storage and indexing approaches, design decisions, and architectures that have been adopted to tackle the various challenges of managing RDF data.

Most of the current RDF systems focus on efficiently executing conjunctive pattern-matching queries. Although such querying constructs represent the backbone of the SPARQL query language, SPARQL allows for much more expressive queries, including transitive closures, optional clauses, and aggregate queries. Efficiently executing such types of queries over massive RDF datasets is an open challenge that still needs to be addressed. In addition, with the high heterogeneity of available RDF datasets, no single set of design decisions or system architecture will ever represent a clear winner for complex SPARQL workloads (Aluç et al. 2014b). Therefore, RDF management systems will increasingly have to incorporate self-tuning capabilities such that they can adapt to dynamic requirements. We believe that these topics will attract significant interest in the near future.

ACKNOWLEDGMENT

The work of Sherif Sakr is funded by the European Regional Development Funds via the Mobilitas Plus programme(grant MOBTT75).

[31] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/.

REFERENCES

Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. 2007. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB Endowment, 411–422.

Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. 2009. HadoopDB: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads. *Proc. VLDB* 2, 1 (2009), 922–933. Retrieved from http://www.vldb.org/pvldb/2/vldb09-861.pdf.

Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. 2011. ANAPSID: An adaptive query processing engine for SPARQL endpoints. *Semant. Web* (2011), 18–34. https://link.springer.com/chapter/10.1007%2F978-3-642-25073-6_2.

Razen Al-Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. 2016. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB J.* 25, 3 (2016), 355–380. DOI: http://dx.doi.org/10.1007/s00778-016-0420-y

Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, and Dimitris Plexousakis. 2001. On storing voluminous RDF descriptions: The case of web portal catalogs. In *Proceedings of the International Workshop on the Web and Databases (WebDB'01)*. 43–48.

Keith Alexander and Michael Hausenblas. 2009. Describing linked datasets—On the design and usage of void, the vocabulary of interlinked datasets. In *Proceedings of the Linked Data on the Web Workshop (LDOW'09)*. Retrieved from http://richard.cyganiak.de/2008/papers/void-ldow2009.pdf.

Güneş Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. 2014a. Diversified stress testing of RDF data management systems. In *Proceedings of the International Semantic Web Conference*. Springer, 197–212.

Güneş Aluç, M. Tamer Özsu, and Khuzaima Daudjee. 2014b. Workload matters: Why RDF databases need a new design. *Proc. VLDB Endow.* 7, 10 (2014), 837–840.

Güneş Aluç, M. Tamer Ozsu, Khuzaima Daudjee, and Olaf Hartig. 2013. *Chameleon-db: A Workload-Aware Robust RDF Data Management System*. Technical Report CS-2013-10. University of Waterloo.

Andrés Aranda-Andújar, Francesca Bugiotti, Jesús Camacho-Rodríguez, Dario Colazzo, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. 2012. AMADA: Web data repositories in the amazon cloud. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM'12)*. 2749–2751. DOI: http://dx.doi.org/10.1145/2396761.2398749

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational data processing in spark. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'15)*. 1383–1394. DOI: http://dx.doi.org/10.1145/2723372.2742797

Medha Atre and James A. Hendler. 2009. BitMat: A main memory bit-matrix of RDF triples. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS'09)*. Citeseer, 33.

Medha Atre, Jagannathan Srinivasan, and James A. Hendler. 2008. BitMat: A main-memory bit matrix of RDF triples for conjunctive triple pattern queries. In *Proceedings of the Poster and Demonstration Session at the 7th International Semantic Web Conference (ISWC'08)*. Retrieved from http://ceur-ws.org/Vol-401/iswc2008pd_submission_16.pdf.

Anirudh Badam and Vivek S. Pai. 2011. SSDAlloc: Hybrid SSD/RAM memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 16–16.

Tim Berners-Lee, James Hendler, Ora Lassila et al. 2001. The semantic web. *Sci. Amer.* 284, 5 (2001), 28–37.

Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using semi-joins to solve relational queries. *J. ACM* 28, 1 (1981), 25–40.

Christian Bizer, Tom Heath, and Tim Berners-Lee. 2009. Linked data-the story so far. https://eprints.soton.ac.uk/271285/.

Christian Bizer and Andreas Schultz. 2009. The Berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.* 5, 2 (2009), 1–24. DOI: http://dx.doi.org/10.4018/jswis.2009040101

Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an efficient RDF store over a relational database. In *Proceedings of the 2013 International Conference on Management of Data*. ACM, 121–132.

Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. 2002. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *Proceedings of the 1st International Semantic Web Conference on the Semantic Web (ISWC'02)*. Springer, 54–68. DOI: http://dx.doi.org/10.1007/3-540-48005-6_7

Rick Cattell. 2011. Scalable SQL and NoSQL data stores. *ACM SIGMOD Rec.* 39, 4 (2011), 12–27.

Surajit Chaudhuri and Gerhard Weikum. 2000. Rethinking database system architecture: Toward a self-tuning RISC-style database system. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB'00)*. 1–10.

Xi Chen, Huajun Chen, Ningyu Zhang, and Songyang Zhang. 2014. SparkRDF: Elastic discreted RDF graph processing engine with distributed memory. In *Proceedings of the Posters & Demonstrations Track a Track Within the 13th International Semantic Web Conference (ISWC'14)*. 261–264. Retrieved from http://ceur-ws.org/Vol-1272/paper_43.pdf.

Xi Chen, Huajun Chen, Ningyu Zhang, and Songyang Zhang. 2015. SparkRDF: Elastic discreted RDF graph processing engine with distributed memory. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT'15)*. 292–300. DOI: http://dx.doi.org/10.1109/WI-IAT.2015.186

Long Cheng and Spyros Kotoulas. 2015. Scale-out processing of large RDF datasets. *IEEE Trans. Big Data* 1, 4 (2015), 138–150. DOI: http://dx.doi.org/10.1109/TBDATA.2015.2505719

Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. 2005. An efficient SQL-based RDF querying scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05)*. VLDB Endowment, 1216–1227. Retrieved from http://portal.acm.org/citation.cfm?id=1083592.1083734.

World Wide Web Consortium. 2014a. RDF 1.1: On Semantics of RDF Datasets. https://www.w3.org/TR/rdf11-datasets/.

World Wide Web Consortium. 2014b. RDF 1.1 Primer.

George P. Copeland and Setrag Khoshafian. 1985. A decomposition storage model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 268–279.

Philippe Cudré-Mauroux, Iliya Enchev, Sever Fundatureanu, Paul Groth, Albert Haque, Andreas Harth, Felix Leif Keppmann, Daniel Miranker, Juan F Sequeda, and Marcin Wylot. 2013. Nosql databases for rdf: An empirical evaluation. In *Proceedings of the International Semantic Web Conference*. Springer, 310–325.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51 (Jan. 2008), 107–113. Issue 1. DOI: http://dx.doi.org/10.1145/1327452.1327492

Gianluca Demartini, Iliya Enchev, Marcin Wylot, Joel Gapany, and Philippe Cudre-Mauroux. 2012. BowlognaBench—Benchmarking RDF analytics. In *Data-Driven Process Discovery and Analysis*, Karl Aberer, Ernesto Damiani, and Tharam Dillon (Eds.). Lecture Notes in Business Information Processing, Vol. 116. Springer, Berlin, 82–102. DOI: http://dx.doi.org/10.1007/978-3-642-34044-4_5

Uwe Deppisch. 1986. S-tree: A dynamic balanced signature index for office retrieval. In *Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 77–87.

Amol Deshpande, Zachary Ives, Vijayshankar Raman et al. 2007. Adaptive query processing. *Foundations and Trends in Databases* 1, 1 (2007), 1–140.

Benjamin Djahandideh, François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. 2015. CliqueSquare in action: Flat plans for massively parallel RDF queries. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE'15)*. 1432–1435. DOI: http://dx.doi.org/10.1109/ICDE.2015.7113394

Orri Erling and Ivan Mikhailov. 2008. Towards web scale RDF. *Proc. SSWS* (2008). https://www.csee.umbc.edu/courses/graduate/691/spring13/01/papers/VOSArticleWebScaleRDF.pdf.

Dieter Fensel. 2003. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer Science & Business Media.

Luis Galárraga, Katja Hose, and Ralf Schenkel. 2014. Partout: A distributed engine for efficient RDF processing. In *23rd International World Wide Web Conference (WWW'14)*. 267–268. DOI: http://dx.doi.org/10.1145/2567948.2577302

José M. Giménez-García, Javier D. Fernández, and Miguel A. Martínez-Prieto. 2015. HDT-MR: A scalable solution for RDF compression with HDT and MapReduce. In *Proceedings of the European Semantic Web Conference*. Springer, 253–268.

François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. 2015. CliqueSquare: Flat plans for massively parallel RDF queries. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE'15)*. 771–782. DOI: http://dx.doi.org/10.1109/ICDE.2015.7113332

Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 599–613. Retrieved from https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez.

Eric L. Goodman and Dirk Grunwald. 2014. Using vertex-centric programming platforms to implement SPARQL queries on large graphs. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms (IA3'14)*. IEEE Press, Piscataway, NJ, 25–32. DOI: http://dx.doi.org/10.1109/IA3.2014.10

Olaf Görlitz and Steffen Staab. 2011. Splendid: Sparql endpoint federation exploiting void descriptions. In *Proceedings of the 2nd International Conference on Consuming Linked Data*. CEUR-WS.org, 13–24.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semant.* 3 (Oct. 2005), 158–182. Issue 2–3. DOI: http://dx.doi.org/10.1016/j.websem.2005.06.005

Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of the International Conference on Management of Data (SIGMOD'14)*. 289–300. DOI: http://dx.doi.org/10.1145/2588555.2610511

Laura Haas, Donald Kossmann, Edward Wimmers, and Jun Yang. 1997. Optimizing queries across diverse data sources. *VLDB*. 276–285. http://www.vldb.org/conf/1997/P276.PDF.

Peter Haase, Katja Hose, Ralf Schenkel, Michael Schmidt, and Andreas Schwarte. 2014. Federated query processing over linked data. In *Linked Data Management*. 369–387. Retrieved from http://www.crcnetbase.com/doi/abs/10.1201/b16859-19.

Peter Haase, Tobias Mathäß, and Michael Ziller. 2010. An evaluation of approaches to federated query processing over linked data. In *Proceedings of the 6th International Conference on Semantic Systems*. ACM, 5.

Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed-Mehdi-Reza Beheshti, and Sherif Sakr. 2015. DREAM: Distributed RDF engine with adaptive query planner and minimal communication. *Proc. VLDB* 8, 6 (2015), 654–665. Retrieved from http://www.vldb.org/pvldb/vol8/p654-Hammoud.pdf.

Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, and Nikos Mamoulis. 2015. Evaluating SPARQL queries on massive RDF datasets. *Proc. VLDB* 8, 12 (2015), 1848–1851. Retrieved from http://www.vldb.org/pvldb/vol8/p1848-harbi.pdf.

Stephen Harris and Nicholas Gibbins. 2003. 3store: Efficient bulk RDF storage. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*. CEUR-WS.org.

Steve Harris, Nick Lamb, and Nigel Shadbolt. 2009. 4store: The design and implementation of a clustered RDF store. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS'09)*. 94–109.

Andreas Harth and Stefan Decker. 2005. Optimized index structures for querying RDF from the web. In *Proceedings of the IEEE Latin American Web Congress (LA-WEB'05)*. 71–80.

Aisha Hasan, Mohammad Hammoud, Reza Nouri, and Sherif Sakr. 2016. DREAM in action: A distributed and adaptive RDF system on the cloud. In *Proceedings of the 25th International Conference on World Wide Web (WWW'16)*. 191–194. DOI: http://dx.doi.org/10.1145/2872518.2901923

Jiewen Huang, Daniel J. Abadi, and Kun Ren. 2011. Scalable SPARQL querying of large RDF graphs. *Proc. VLDB* 4, 11 (2011), 1123–1134.

Mohammad Husain, James McGlothlin, Mohammad M. Masud, Latifur Khan, and Bhavani M. Thuraisingham. 2011. Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Trans. Knowl. Data Eng.* 23, 9 (2011), 1312–1327.

Vijay Ingalalli, Dino Ienco, Pascal Poncelet, and Serena Villata. 2016. Querying RDF data using a multigraph-based approach. In *Proceedings of the 19th International Conference on Extending Database Technology (EDBT'16)*. 245–256. DOI: http://dx.doi.org/10.5441/002/edbt.2016.24

Zoi Kaoudi and Ioana Manolescu. 2015. RDF in the clouds: A survey. *VLDB J.* 24, 1 (2015), 67–91.

Vaibhav Khadilkar, Murat Kantarcioglu, Bhavani M. Thuraisingham, and Paolo Castagna. 2012. Jena-HBase: A distributed, scalable and effcient RDF triple store. In *Proceedings of the ISWC 2012 Posters & Demonstrations Track*. Retrieved from http://ceur-ws.org/Vol-914/paper_14.pdf.

HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. 2013. Optimizing RDF(S) queries on cloud platforms. In *Proceedings of the 22nd International World Wide Web Conference (WWW'13)*. 261–264. Retrieved from http://dl.acm.org/citation.cfm?id=2487917.

Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming subgraph isomorphism for RDF query processing. *Proc. VLDB* 8, 11 (2015), 1238–1249. Retrieved from http://www.vldb.org/pvldb/vol8/p1238-kim.pdf.

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Vol. 8. 31–46.

Günter Ladwig and Andreas Harth. 2011. CumulusRDF: Linked data management on nested key-value stores. In *Proceedings of the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS'11)*. 30.

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. DOI: http://dx.doi.org/10.1145/1773912.1773922

Kisung Lee and Ling Liu. 2013. Scaling queries over big RDF graphs with semantic hash partitioning. *Proc. VLDB Endow.* 6, 14 (2013), 1894–1905.

Baolin Liu and Bo Hu. 2005. An evaluation of RDF storage systems for large data applications. In *Proceedings of the 1st International Conference on Semantics, Knowledge and Grid*. IEEE, 59–59.

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed graphlab: A framework for machine learning in the cloud. *Proc. VLDB* 5, 8 (2012), 716–727. Retrieved from http://vldb.org/pvldb/vol5/p716_yuchenglow_vldb2012.pdf.

Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao Liu. 2004. RStar: An RDF storage and query system for enterprise resource management. In *Proceedings of the 13th ACM International Conference on Information and Knowledge Management*. ACM, 484–491.

Miguel A. Martínez-Prieto, Mario Arias, and Javier D. Fernandez. 2012. Exchange and consumption of huge RDF data. In *The Semantic Web: Research and Applications*. Springer, 437–452.

Brian McBride. 2002. Jena: A semantic web toolkit. *IEEE Internet Comput.* 6, 6 (2002), 55–59.

Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. 2011. DBpedia SPARQL benchmark–Performance assessment with real queries on real data. In *Proceedings of the International Semantic Web Conference (ISWC'11)*. Springer, 454–469.

Raghava Mutharaju, Sherif Sakr, Alessandra Sala, and Pascal Hitzler. 2013. D-SPARQ: Distributed, scalable and efficient RDF query engine. In *Proceedings of the ISWC 2013 Posters & Demonstrations Track*. 261–264. Retrieved from http://ceur-ws.org/Vol-1035/iswc2013_poster_21.pdf.

Hubert Naacke, Olivier Curé, and Bernd Amann. 2016. SPARQL query processing with apache spark. *CoRR* abs/1604.08903 (2016). Retrieved from http://arxiv.org/abs/1604.08903.

Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: A RISC-style engine for RDF. *Proc. VLDB Endow.* 1, 1 (2008), 647–659.

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

Andriy Nikolov, Andreas Schwarte, and Christian Hütter. 2013. Fedsearch: Efficiently combining structured queries and full-text search in a SPARQL federation. In *Proceedings of the International Semantic Web Conference*. Springer, 427–443.

Damla Oguz, Belgin Ergenc, Shaoyi Yin, Oguz Dikenelli, and Abdelkader Hameurlain. 2015. Federated query processing on linked data: A qualitative survey and open challenges. *Knowl. Eng. Rev.* 30, 5 (2015), 545–563.

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. 1099–1110. DOI: http://dx.doi.org/10.1145/1376616.1376726

M. Tamer Özsu. 2016. A survey of RDF data management systems. *Front. Comput. Sci.* 10, 3 (2016), 418–432.

Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. 2013. H2RDF+: High-performance distributed joins over large-scale RDF graphs. In *Proceedings of the 2013 IEEE International Conference on Big Data*. 255–263. DOI: http://dx.doi.org/10.1109/BigData.2013.6691582

Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. 2012. H2RDF: Adaptive query processing on RDF data in the cloud. In *Proceedings of the 21st World Wide Web Conference (WWW'12)*. 397–400. DOI: http://dx.doi.org/10.1145/2187980.2188058

Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras, and Nectarios Koziris. 2014. H_2RDF+: An efficient data management system for big RDF graphs. In *Proceedings of the International Conference on Management of Data (SIGMOD'14)*. 909–912. DOI: http://dx.doi.org/10.1145/2588555.2594535

Peng Peng, Lei Zou, Lei Chen, and Dongyan Zhao. 2016. Query workload-based RDF graph fragmentation and allocation. In *Proceedings of the 19th International Conference on Extending Database Technology (EDBT'16)*. 377–388. DOI: http://dx.doi.org/10.5441/002/edbt.2016.35

Minh-Duc Pham, Peter Boncz, and Orri Erling. 2012. S3g2: A scalable structure-correlated social graph generator. In *Proceedings of the Technology Conference on Performance Evaluation and Benchmarking*. Springer, 156–172.

Roshan Punnoose, Adina Crainiceanu, and David Rapp. 2015. SPARQL in the cloud using Rya. *Inf. Syst.* 48 (2015), 181–195. DOI: http://dx.doi.org/10.1016/j.is.2013.07.001

Nur Aini Rakhmawati, Jürgen Umbrich, Marcel Karnstedt, Ali Hasnain, and Michael Hausenblas. 2013. Querying over federated SPARQL endpoints—A state of the art survey. *arXiv Preprint arXiv:1306.1723* (2013).

Louiqa Raschid and Stanley Y. W. Su. 1986. A parallel processing strategy for evaluating recursive queries. In *Proceedings of the Conference on Very Large Data Bases (VLDB'86)*, Vol. 86. 412–419.

Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. 2011. An intermediate algebra for optimizing RDF graph pattern matching on mapreduce. In *Proceedings of the 8th Extended Semantic Web Conference: Research and Applications (ESWC'11)*. 46–61. DOI: http://dx.doi.org/10.1007/978-3-642-21064-8_4

Kurt Rohloff and Richard E. Schantz. 2010. High-performance, massively scalable distributed systems using the mapreduce software framework: The SHARD triple-store. In *Programming Support Innovations for Emerging Distributed Applications*. ACM, 4.

Sherif Sakr, Anna Liu, Daniel M. Batista, and Mohammad Alomari. 2011. A survey of large scale data management approaches in cloud environments. *IEEE Commun. Surveys Tutor.* 13, 3 (2011), 311–336. DOI: http://dx.doi.org/10.1109/SURV.2011.032211.00087

Sherif Sakr, Anna Liu, and Ayman G. Fayoumi. 2013. The family of mapreduce and large-scale data processing systems. *Comput. Surveys* 46, 1 (2013).

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

Alexander Schätzle, Martin Przyjaciel-Zablocki, Thorsten Berberich, and Georg Lausen. 2015. S2X: Graph-parallel querying of RDF with graphX. In *Proceedings of the 1st International Workshop on Big-Graphs Online Querying (BigOQ'15)*.

Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, and Georg Lausen. 2013. PigSPARQL: A SPARQL query processing baseline for big data. In *Proceedings of the ISWC 2013 Posters & Demonstrations Track*. 241–244. Retrieved from http://ceur-ws.org/Vol-1035/iswc2013_poster_16.pdf.

Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. 2015. S2RDF: RDF querying with SPARQL on spark. *CoRR* abs/1512.07021 (2015). Retrieved from http://arxiv.org/abs/1512.07021.

M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel. 2008. An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario. In *Proceedings of the International Semantic Web Conference (ISWC'08)*. 82–97.

M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. 2009. SPˆ 2bench: A SPARQL performance benchmark. In *Proceedings of the IEEE 25th International Conference on Data Engineering (ICDE'09)*. IEEE, 222–233.

Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. 2011. Fedx: Optimization techniques for federated query processing on linked data. In *Proceedings of the International Semantic Web Conference*. Springer, 601–616.

Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 International Conference on Management of Data*. ACM, 505–516.

Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. 2008. Column-store support for RDF data management: Not all swans are white. *Proc. VLDB Endow.* 1, 2 (2008), 1553–1563.

Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. 2008. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th International Conference on World Wide Web (WWW'08)*. ACM, 595–604.

M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. 2005. C-store: A column oriented DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'05)*.

Philip Stutz, Abraham Bernstein, and William Cohen. 2010. Signal/collect: Graph algorithms for the (semantic) web. In *Proceedings of the International Semantic Web Conference*. Springer, 764–780.

Philip Stutz, Bibek Paudel, Mihaela Verman, and Abraham Bernstein. 2015. Random walk triplerush: Asynchronous graph querying and sampling. In *Proceedings of the 24th International Conference on World Wide Web (WWW'15)*. ACM, 1034–1044.

Tolga Urhan and Michael J. Franklin. 2000. Xjoin: A reactively scheduled pipelined join operatorỳ. *Bull. Tech. Committee* (2000), 27.

Patrick Valduriez. 1987. Join indices. *ACM Trans. Database Syst.* 12, 2 (1987), 218–246. DOI: http://dx.doi.org/10.1145/22952.22955

Xin Wang, Thanassis Tiropanis, and Hugh C. Davis. 2013. Lhd: Optimising linked data query processing using parallelisation. *LDOW*. http://ceur-ws.org/Vol-996/papers/ldow2013-paper-06.pdf.

Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endow.* 1, 1 (2008), 1008–1019. DOI: http://dx.doi.org/10.1145/1453856.1453965

Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. 2003. Efficient RDF storage and retrieval in jena2. In *Proceedings of the International Conference on Semantic Web and Databases (SWDB'03)*. 131–150.

Kevin Wilkinson and Kevin Wilkinson. 2006. Jena property table implementation. In *Proceedings of the International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS'06)*.

Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Hai Jin, and Ling Liu. 2014. SemStore: A semantic-preserving distributed RDF triple store. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM'14)*. 509–518. DOI: http://dx.doi.org/10.1145/2661829.2661876

Marcin Wylot and Philippe Cudré-Mauroux. 2016. DiploCloud: Efficient and scalable management of RDF data in the cloud. *IEEE Trans. Knowl. Data Eng.* 28, 3 (2016), 659–674. DOI: http://dx.doi.org/10.1109/TKDE.2015.2499202

Marcin Wylot, Jigé Pont, Mariusz Wisniewski, and Philippe Cudré-Mauroux. 2011. dipLODocus[RDF] - Short and long-tail RDF analytics for massive webs of data. In *Proceedings of the International Semantic Web Conference*. 778–793.

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

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'10)*. Retrieved from https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.

Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A distributed graph engine for web scale RDF data. In *Proceedings of the 39th International Conference on Very Large Data Bases*. VLDB Endowment, 265–276.

Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. 2013. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE'13)*. 565–576. DOI: http://dx.doi.org/10.1109/ICDE.2013.6544856

Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. 2014. gStore: A graph-based SPARQL query engine. *VLDB J.* 23, 4 (2014), 565–590. DOI: http://dx.doi.org/10.1007/s00778-013-0337-7

Received November 2016; revised December 2017; accepted December 2017