RDF in the Clouds: A Survey

Zoi Kaoudi, Ioana Manolescu

HAL Id: hal-01020977

https://hal.inria.fr/hal-01020977

Submitted on 8 Jul 2014

Noname manuscript No.
(will be inserted by the editor)

RDF in the Clouds: A Survey

Zoi Kaoudi · Ioana Manolescu

Received: date / Accepted: date

Abstract The Resource Description Framework (RDF) pioneered by the W3C is increasingly being adopted to model data in a variety of scenarios, in particular data to be published or exchanged on the Web. Managing large volumes of RDF data is challenging, due to the sheer size, the heterogeneity, and the further complexity brought by RDF reasoning. To tackle the size challenge, distributed storage architectures are required. Cloud computing is an emerging paradigm massively adopted in many applications for the scalability, fault-tolerance and elasticity features it provides, enabling the easy deployment of distributed and parallel architectures. In this article, we survey RDF data management architectures and systems designed for a cloud environment, and more generally, those large-scale RDF data management systems that can be easily deployed therein. We first give the necessary background, then describe the existing systems and proposals in this area, and classify them according to dimensions related to their capabilities and implementation techniques. The survey ends with a discussion of open problems and perspectives.

1 Introduction

The Resource Description Framework (RDF) pioneered by the W3C [42] is increasingly being adopted to model data in a variety of scenarios, in particular data to be published or exchanged on the Web. An RDF dataset consists of *triples* stating that a *resource* has a *property* with a certain *value*. Further, RDF comes endowed with an associated schema (ontology) language, namely RDF Schema (or RDFS, in short), for describing *classes* to which resources belong, resource *properties*, and the *relationships* which hold between such classes and properties [19]. For instance, using RDF one can state that any *student* is also a *human*, or that if X *worksWith* Y, then X also *knows* Y; or that if X *drives* car Z, then X is a *human* etc. These examples show that RDFS statements lead to *entailed* (or derived) facts. When an RDF Schema is available for a dataset, RDF semantics requires considering that the database consists not only of triples explicitly present in the store, but also of a set of *entailed triples* obtained through *reasoning based on an RDF Schema and the RDFS entailment rules*.

A large class of interesting RDF applications comes from the Open Data concept that "*certain data should be freely available to everyone to use and republish as they wish, without restrictions from copyright, patents or other mechanisms of control*"[1]. Open Data federates players of many roles, from organizations such as business and government aiming at demonstrate transparency and good (corporate) governance, to end users interested in consuming and producing data to share with the others, to aggregators that may build business models around warehousing, curating, and sharing this data [69]. Sample governmental Open Data portals are the ones from the US (`www.data.gov`), UK (`www.data.gov.uk`) and France (`www.etalab.fr`). While Open Data designates a general philosophy, Linked Data refers to the "*recommended best practice for exposing, sharing, and connecting pieces of data, information, and knowledge on the Semantic Web using URIs and RDF*" [16]. In practice, Open and Linked data are frequently combined to facilitate data sharing, interpretation, and exploitation [56].

Z. Kaoudi
Inria Saclay–Île-de-France and Université Paris-Sud
Bâtiment 650 (PCRI), 91405 Orsay Cedex, France
E-mail: zoi.kaoudi@inria.fr
Present address: IMIS, Athena Research Center, Greece

I. Manolescu
Inria Saclay–Île-de-France and Université Paris-Sud
Bâtiment 650 (PCRI), 91405 Orsay Cedex, France
E-mail: ioana.manolescu@inria.fr

[1] `http://en.wikipedia.org/wiki/Open_data`

Sample applications of Linked Open Data are DBPedia (the Linked Data version of Wikipedia), BBC's platform for the World Cup 2010 and the 2012 Olympic games [53]. Interesting RDF datasets may involve large volumes of data. For instance, `data.gov` comprises more than 5 billion triples, while the latest version of DBPedia corresponds to more than 2 billion triples. Many large datasets have been also gathered in life sciences for integrating and analyzing large biomedical datasets [74], some of them listed at `www.w3.org/wiki/DataSetRDFDumps`. For example, the Linked Cancer Genome Atlas dataset currently consists of 7.36 billion triples and is estimated to reach 30 billions [79]. Other large RDF projects, involving billions of triples, are described in [63] (mostly projects carried by the Ontotext company).

To store and query RDF data, many systems have been built, firstly within the Semantic Web community such as Jena [89] and Sesame [20]. RDF storage, indexing and query processing has also attracted interest from the data management community [1, 60, 88] and lately, commercial database management systems also started providing support for RDF, such as Oracle 11g [25] or IBM DB2 10.1 [18]. These works mostly focus on the evaluation of conjunctive queries on RDF databases, and do not consider RDF-specific features such as those related to reasoning.

The need to scale beyond the capacity of a single-site server has brought about distributed RDF management systems. In particular, since early interest in RDF coincided with important research invested into peer-to-peer data management platforms, peer-to-peer RDF data management algorithms were proposed for instance in [49, 50]. Peer-to-peer platforms sought to capitalize on the availability of heterogeneous, distributed hardware owned by numerous users to implement large-scale distributed applications and in particular data sharing. They placed a strong emphasis in preserving some level of peer independence, and adapting gracefully to peers and/or datasets joining, respectively, leaving the network, which in turn complicates the application's resilience to failures, and maintaining some level of service. A survey on P2P-based RDF data management can be found in [36].

More recently, in particular since the advent of inexpensive distributed hardware, research in distributed data management has focused on platforms with centralized control where one or a few master nodes organize the work of many "slave" ones; the MapReduce [28] distributed programming model exemplifies this mindset, with a master distributing and supervising work between the other nodes (slaves) of a cluster. In particular, cluster-based RDF stores have been developed, such as 4store [40], the clustered version of Jena [64] or Virtuoso [35]. Finally, the term *cloud computing* is used to designate a broad family of distributed storage and processing architectures, whose main common characteristics are: scalability, fault-tolerance, and elastic allocation of as much storage and computing power as needed at a particular point in time by each application. Another valued feature of most current cloud computing platforms is that they release the application owner or developer from the burden of administering hardware and software resources and simplify the deployment of large-scale distributed applications. These features have led the Semantic Web and data management community to investigate a variety of architectures, and develop platforms, for the cloud-based management of large volumes of RDF data.

The goal of our work is to provide a comprehensive survey of *RDF data management in cloud environments*, or, more broadly speaking, large-scale distributed platforms, where storage and query processing are performed in a distributed fashion but under a centralized control (as opposed to decentralized peer-to-peer systems). Our choice of focus is motivated first, by the wide current availability of commercial and private cloud platforms, and second, by the shared goals of RDF platforms developed either for the cloud or simply for a large-scale cluster: being able to store and process efficiently very large volumes of Semantic Web data, in an architecture with a single point of control over the distributed nodes.

We provide a description of existing systems and proposals in the area of our survey, and classify them according to different dimensions related to their capabilities and implementation techniques. Our first identified dimensions refer to the fundamental functionalities of RDF stores: *data storage*, *query processing*, and *reasoning*. Then, within each dimension we highlight the aspects of centralized RDF data management that need to be revised in a cloud environment and classify each system according to their basic characteristics. Finally, we provide a high-level view of the systems classification along these dimensions, pointing out the areas where numerous works compete and those where further progress is needed and unexplored paths remain.

The rest of the paper is organized as follows. We start in Section 2 by introducing the main features of RDF and its accompanying schema language RDFS, while Section 3 gives an overview of the cloud-based frameworks and tools used for RDF data management. In Section 4 we present current approaches on RDF data storage, in Section 5 we describe different query processing paradigms for evaluating RDF queries and in Section 6 we layout the state-of-the-art in RDF reasoning on top of cloud platforms. Section 7 gives a high-level comparison and classification of existing systems. Finally, we conclude in Section 8 and give insights into open problems and directions.

2 The RDF data model and its query language

We briefly recall the main features of the RDF data model [54, 57] and its query language, SPARQL [41, 67].

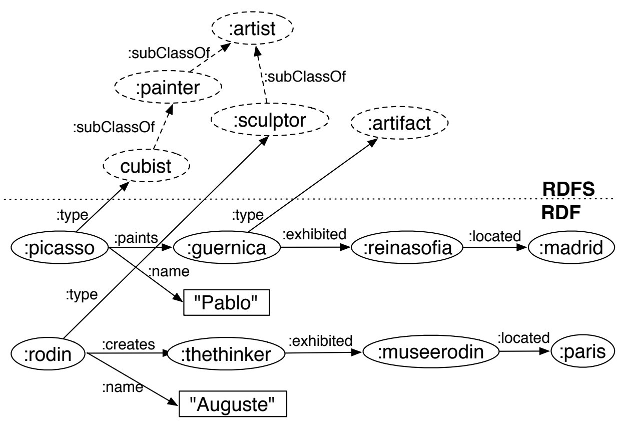


Fig. 1: RDF(S) graph.

```
:sculptor :subClassOf :artist .
:painter :subClassOf :artist .
:cubist :subClassOf :painter.
:picasso :type :cubist .
:picasso :name "Pablo" .
:picasso :paints :guernica .
:guernica :type :artifact .
:guernica :exhibited :reinasofia .
:reinasofia :located :madrid .
:rodin :type :sculptor .
:rodin :name "Auguste" .
:rodin :creates :thethinker .
:thethinker :exhibited :museerodin .
:museerodin :located :paris .
```

Fig. 2: RDF(S) graph of Fig. 1 in N-Triples syntax.

2.1 RDF and RDF Schema

RDF data is organized in graphs consisting of *triples* of the form (s, p, o), stating that the subject node s has the property edge p whose value is the object node o. A key concept for RDF is that of URIs or Unique Resource Identifiers; these can be used in either of the s, p and o positions to uniquely refer to some entity, relationship or concept. Literals (constants) are also allowed in the o position.

RDF allows representing a form of incomplete information [2] through *blank nodes*, standing for unknown constants or URIs; an RDF database may, for instance, state that the *author* of X is *Jane* while the *date* of X is *4/1/2011*, for a given, unknown resource X. A query on this database asking for what Jane wrote on 4/1/2011 should return X; in other terms, joins are possible on blank nodes (which can be alternatively viewed as "labeled nulls"). This contrasts with standard relational databases where all attribute values are either constants or $null$ (and $null$ never compares equal to any other constant, neither to $null$).

Formally, let U, L and B denote three pairwise disjoint sets of URIs, literals, and blank nodes, respectively. A *triple* is a tuple (s, p, o) from $(U \cup B) \times U \times (U \cup L \cup B)$, where s is the subject, p is the property (a.k.a. predicate) and o is the object of the triple. In the following, we will refer to an RDF *term* as any value of $U \cup L \cup B$ and to an RDF *element* as any among the subject, predicate, and object of an RDF triple. In this paper, we attach the prefix '`:`' for all URIs to distinguish them from the corresponding literals.

An RDF graph (or dataset) is a set of triples. It encodes a graph structure in which every triple (s, p, o) describes a directed edge labelled with p from the node labelled with s to the node labelled with o. There is a variety of syntaxes for RDF (e.g., RDF/XML, N-Triples, Turtle, etc.). The most basic one is N-Triples, which contains one triple per line.

RDFS [19] is the accompanying W3C proposal of a schema language for RDF. It is used to describe *classes* and relationships between classes (such as inheritance). Further, it allows specifying *properties*, and relationships that may hold between pairs of properties, or between a class and a property. Given that any class (or property) is a resource in itself, RDFS statements are represented by triples; thus an RDFS can be seen as an RDF graph in itself. We call an RDFS triple *schema triple*, and any other triple a *data triple*. In the RDF and RDFS world, to state that a resource r is of a type τ, a triple of the form "r `:type` τ" is used. Since this triple is about the resource r (not about the class τ), it is viewed as a data triple.

Figure 1 shows an illustration of an RDF(S)[2] graph from the cultural domain. The upper level of the graph depicts the schema, while the lower level is the RDF data. Resources are oval-shaped, with RDFS classes shown in dashed ovals. Rectangles denote literals which in our example are strings; an arrow outgoing from a resource node describes a property of that resource, with the property name being the label on the arrow. The graph of Figure 1 can also be represented by a set of RDF triples as shown in Figure 2 in the N-Triples syntax.

A distinguishing feature of RDF and RDFS is the ability to infer new RDF triples (*entailed triples*) based on a set of RDFS entailment rules [42]. For instance, in the example of Figure 1, we can infer that `:rodin` is also of the type `:artist`, because `:rodin` is an instance of `:sculptor` and `:sculptor` is a subclass of `:artist`. Then, if a query asks for the instances of class `:artist`, the answer should contain `:rodin` although it is not explicitly present in the data. This process is often referred to as *RDFS entailment* or *reasoning*.

The complete set of RDFS entailment rules proposed by W3C consists of thirteen rules and can be found in [42]. Some of these rules are crucial and occur frequently while modelling an application domain; others mostly capture the internals of RDFS, stating, for instance, that any URI ele-

[2] From now on, we will use the term RDF(S) to refer to both RDF and RDFS.

Table 1: Minimal RDFS entailment rules.

Rule	Triples	Entailed triple
s_1	(c_1 :subClassOf c_2), (c_2 :subClassOf c_3)	(c_1 :subClassOf c_3)
s_2	(p_1 :subPropertyOf p_2), (p_2 :subPropertyOf p_3)	(p_1 :subPropertyOf p_3)
i_1	(p_1 :subPropertyOf p_2), (s p_1 o)	(s p_2 o)
i_2	(c_1 :subClassOf c_2), (s :type c_1)	(s :type c_2)
i_3	(p :domain c), (s p o)	(s :type c)
i_4	(p :range c), (s p o)	(o :type c)

```
:picasso :type :painter .
:picasso :type :artist .
:rodin :type :artist .
:cubist :subClassOf :artist .
```

Fig. 3: Entailed triples of RDF(S) graph of Fig. 1.

```
SELECT ?y ?z
WHERE {
        ?x :type :artist .
        ?x :paints ?y .
        ?y :exhibited ?z .
        ?z :located :paris . }
```

Fig. 4: Example SPARQL query.

ment is an instance of class :Resource. In [59] the authors present a small subset of the RDFS entailment rules which preserves the core functionalities and is sound and complete. This fragment is called *minimal RDFS* in [59] and its rules are shown in Table 1. This set of rules is the one largely used by RDF data management systems that support reasoning.

Figure 3 shows all the triples entailed from the RDF graph of Figure 1 based on the above set of rules. Taking these triples into account together with the triples of Figure 2, the answer of the query that asks for all the instances of :artist includes :rodin and :picasso, although these were not directly declared of type :artist in the original RDF graph.

If more expressive schema or constraints are needed, going beyond RDFS, one could use ontology languages such as OWL [39] to describe the properties of RDF databases. One commonly supported OWL fragment is the OWL Horst fragment consisting of 24 rules [80].

2.2 SPARQL

For what concerns RDF querying, SPARQL [41] is the W3C standard for querying RDF graphs. A commonly-used subset of SPARQL is the Basic Graph Pattern (BGP) queries of SPARQL, i.e., its conjunctive fragment allowing to express the core Select-Project-Join database queries. In such queries, the notion of triple is generalized to that of *triple pattern* (s, p, o) from $(U \cup B \cup V) \times (U \cup V) \times (U \cup L \cup B \cup V)$, where V is a set of variables. The normative syntax of BGP queries is:

$$\texttt{SELECT } ?v_1 \ldots ?v_m \texttt{ WHERE } \{t_1 \ldots t_n\}$$

where $t_1, \ldots, t_n$ is a set of triple patterns, and $?v_1 \ldots ?v_m$ a set of variables occurring in $\{t_1 \ldots t_n\}$ that defines the output of the query (distinguished variables). In the above, repeated use of a variable encodes a join. Blank nodes can also appear in a triple pattern of a query, and are treated as non-distinguished variables.

Two common types of BGP queries are *star-join* queries, queries of k triple patterns containing the same variable in the subject position, and *path or chain* queries, queries of k triple patterns where the object of each triple pattern is joined with the subject of the next one.

A sample conjunctive SPARQL query, which we will use throughout the paper, is shown in Figure 4. The query asks for the artists' paintings which are exhibited in museums located in Paris. The first two triple patterns are joined on variable ?x (forming a star-join subquery), the second with the third on variable ?y and the last two on variable ?z. The last three triple patterns form a path subquery.

BGP query semantics can be briefly outlined as follows. First, a mapping μ from $B \cup V$ to $U \cup B \cup L$ is defined as a partial function $\mu : B \cup V \rightarrow U \cup B \cup L$. If t is a triple pattern, t^{μ} denotes the result of replacing the blank nodes and variables in t according to μ. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined. Let q = $\texttt{SELECT } ?v_1 \ldots ?v_m \texttt{ WHERE } \{t_1 \ldots t_n\}$ be a BGP query and G the RDF graph against which q should be evaluated. The evaluation of q is: $eval(q) = \{\mu(?v_1 \ldots ?v_m) \mid dom(\mu) = varbl(q) \ and \ \{t_1^{\mu}, t_2^{\mu}, ..., t_n^{\mu}\} \subseteq G\}$, with $varbl(q)$ the set of variables and blank nodes occurring in q.

To obtain complete query results, all the results based also on entailed triples must be produced. Let G_{rdfs} denote the original RDF graph to which all the entailed triples have been added. The result of evaluating a query q on G_{rdfs} is then: $eval_{rdfs}(q) = \{\mu(?v_1 \ldots ?v_m) \mid dom(\mu) = varbl(q) \ and \ \{t_1^{\mu}, t_2^{\mu}, ..., t_n^{\mu}\} \subseteq G_{rdfs}\}$

SPARQL also supports more complex queries than simple BGPs with features such as the OPTIONAL operation (a triple pattern can be optionally matched), filter expressions with operators such as $<$, $=$, and $>$ for numerical values (range constraints) and string functions such as regular expressions. Multiple BGPs can be combined through a UNION operation. Order constraints are also possible through ORDERBY and LIMIT operators. The latest SPARQL 1.1 proposal [41] also supports property paths, subqueries, negation, aggregates etc. We do not consider these features in the

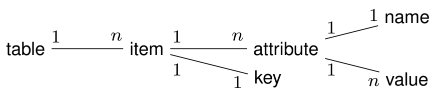


Fig. 5: Structure of a table in a key-value store.

present survey, and instead focus on the core BGP dialect, which is by far the most widely supported in the existing platforms.

3 Building blocks: key-value stores and MapReduce systems

Existing cloud- and/or massively distributed RDF stores have relied significantly on novel software systems and platforms, in particular on NoSQL key-value stores, and on MapReduce implementations such as Hadoop. To make our presentation self-contained, we briefly review here these concepts and the associated notations. The interested reader may find in [23] a survey of NoSQL systems while [32, 73] cover MapReduce variations and other massively parallel data management frameworks.

3.1 Key-value stores

An important family of NoSQL systems are the distributed key-value stores, providing very simple data structures based on the concept of (key, value) pairs. Such stores typically handle *items*, each of which consist of a *key* and several *attributes*; in turn, an attribute consists of a *name* and one or several *values*. For convenience, most key-value stores also support *named collections* of items, which are typically called *tables*. Figure 5 outlines this simple data model.

Key-value stores are notably *schema-less*, in the sense that there can be any set of attributes associated to each key, while each attribute name can have multiple values. Key-value stores offer a simple interface based on the operations PUT$(table, key, items)$ and GET$(table, key)$, and do not support operations requiring data from different tables. Therefore, if data needs to be combined across tables, this must take place within the application layer.

Popular key-value stores include Apache Accumulo [7], Amazon's DynamoDB [13], and Apache HBase [10]. Although they share the basic elements of their interfaces, these systems differ with respect to their internal architecture (client-server vs. P2P-based), access control policies, authentication, consistency etc. One difference that affects the design of an RDF store relying on such platforms is whether the index offered on the key is hash-based (allowing only direct lookups), or sorted (which furthermore allows for prefix lookups).

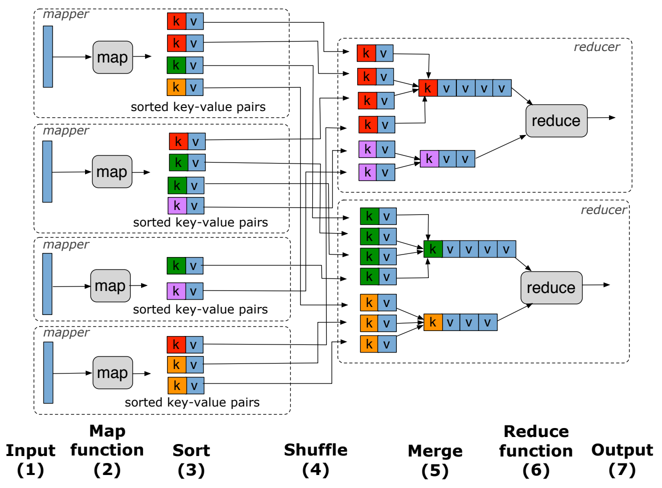


Fig. 6: MapReduce functionality.

Some NoSQL systems support a more complex, nested data model, usually referred to as *extended key-value stores*. In such stores, multiple attribute name-value pairs of a certain key can be nested into *super attributes* (a.k.a. super columns). Examples of extended key-value stores include Google's BigTable [24] and Apache's Cassandra [8]. In Cassandra [8], a sorted index is available on both attributes and super attributes, as well as a secondary index that maps attribute values to keys.

3.2 MapReduce and Hadoop

Interest in massively parallel processing has been renewed recently since the emergence of the MapReduce framework [28] and its open source implementation Hadoop [9]. MapReduce has become popular in various computer-science fields as it provides a simple programming paradigm which frees the developer from the burden of handling parallelization, scalability, load balancing and fault-tolerance.

MapReduce processing is organized in *jobs*. In turn, each job consists of a *map* and a *reduce* phase, separated by a *shuffle* (data transfer) phase. The map phase is specified by the user-defined function MAP which takes as input $(key, value)$ pairs, performs some tasks on these (if needed) and outputs intermediate pairs $(ikey, ivalue)$. These pairs are shuffled through the network and are given as input to the reduce phase. The reduce phase is specified by the user-defined function REDUCE which takes as an input all the $ivalues$ that have the same $ikey$. The user can also specify the shuffle phase by defining the PARTITION function.

The file system splits data into chunks either automatically to a default size or in a way specified by the user. The key aspect of MapReduce enabling parallelism is that a map (respectively, a reduce) phase is performed by several *mappers* (respectively *reducers*), each of which are independent processes operating on separate chunks of data. The nodes of the cluster run in parallel one or more map/reduce tasks.

Figure 6 illustrates how MapReduce works. Each input data chunk initializes a mapper task (1), which processes the input according to the user-defined MAP function (2). The function outputs intermediate key-value pairs, which are then sorted according to their key (3). The intermediate pairs are shuffled and the ones with the same key (same color in the figure) are routed to the same reducer (4). In the reducer, the values corresponding to the same key are merged into a list (5). Finally, the REDUCE function operates on the list of values for a specific key (6) and outputs the results (7).

Hadoop [9] is the most popular open-source implementation of Google's MapReduce engine. HDFS is Hadoop's distributed file system, analogous to Google File System built for Google's MapReduce engine. HDFS purpose is to store very large files in a distributed and robust fashion. In particular, it stores data in blocks of constant size (64 MB by default) which are replicated within the system (3 times by default).

Although MapReduce was first mostly intended for data analysis tasks, it has also started being used in query processing tasks. However, MapReduce does not directly support more complex operations such as joins. Techniques for MapReduce-based join evaluation are proposed in [5, 6, 17, 32]. Two of the most widely-used ones is the *standard repartition join* and the *broadcast join*. The former distributes the two relations on the join key during the shuffle phase and joins them in the reduce phase. The latter broadcasts the smaller relation to all nodes and performs the join at the map phase.

4 Cloud-based RDF storage

This section focuses on the storage subsystems of the existing RDF cloud data management systems. From the angle of their underlying stores, existing systems can be classified into the following categories:

- systems relying on a distributed file system, such as HDFS;
- systems which use existing "NoSQL" key-value stores [23] as back-ends;
- systems warehousing RDF data in a "federation" of single-site (centralized) RDF stores, one on each node;
- hybrid systems using a combination of the above.

The questions to be answered by a large-scale distributed RDF storage platform are: how to partition the data across nodes, and how to store on each node the corresponding partition. Figure 7 proposes a classification of existing RDF stores according to the back-ends and partitioning schemes they rely on; this figure will serve as a basis of our discussion.

4.1 Storing RDF data in a distributed file system

Distributed file systems (DFS) are designed for scalable and reliable data storage in a cluster of commodity machines and, thus, they are a good fit for storing large files of RDF data in the cloud. Large files are split into chunks, distributed among the cluster machines and automatically replicated by the DFS for fault-tolerance reasons. The server of the DFS is usually the node responsible for replicating and keeping track of the different chunks of data making up each file. Distributed file systems do not provide fine-grained data access, and thus, selective access to a piece of data can only be achieved through a full scan of all the files. Recent works such as [30, 31] are based on Hadoop and improve its data access efficiency with indexing functionalities. However, these techniques have not yet been fully adopted within the Hadoop development community.

We classify the systems using DFS to store RDF data according to the way they model the data as follows: (i) the triple model which preserves the triple construct of RDF, (ii) the vertical partitioning model which splits RDF triples based on their property, and (iii) the entity class-based model which uses high-level entity classes graph to partition the RDF triples.

4.1.1 Triple model

The simplest way to store RDF data into a DFS is by loading each file into the DFS. The file system is then in charge of: splitting the files into blocks, replicating them and placing them at the cluster nodes. Conceptually this can be considered as storing all triples in a 3-attribute relational table that has no indices in a centralized environment.

Systems based on this approach include [71, 75] which use the Hadoop Distributed File System (HDFS). In SHARD [71] a slight variation is used where triples having the same subject are grouped in one line in the file. For instance, for the subject `:picasso` in our running example of Figure 1 we would have the following line in a file in HDFS:
`:picasso :paints :guernica :name "Pablo" :type :cubist.`

Neither of the works in this category explicitly determine how triples are grouped in files; this task is left to the application or user.

4.1.2 Vertical partitioning model

A more elaborate way to store RDF data into a DFS is to partition it into smaller files and thus, enable finer-granularity access to the data. To this end, RDF triples are partitioned based on the value of their property and each partition is stored within one file into the DFS, named according to the respective property value. As the property can explicitly be inferred from the file name, the property-partitioned files

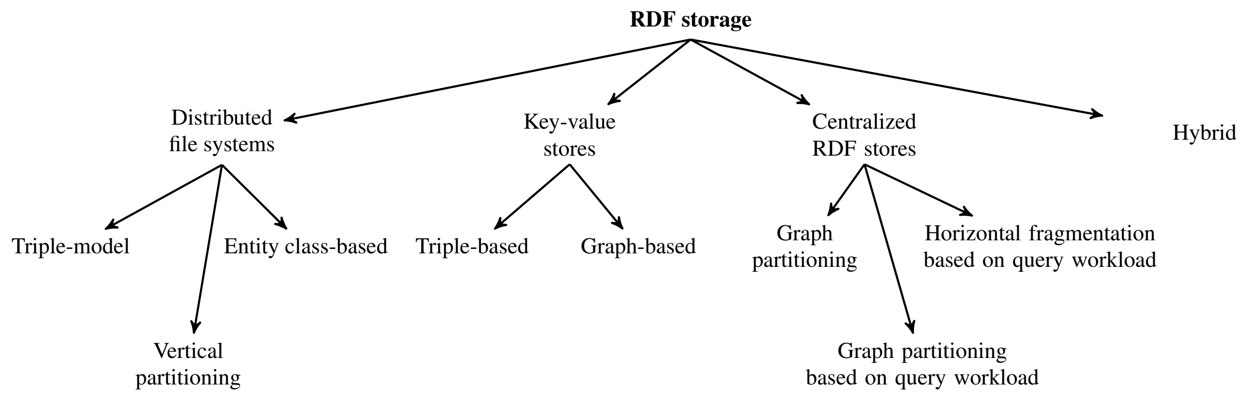


Fig. 7: Taxonomy of storage back-ends and partitioning schemes used by the systems.

need only to store the subject and object of each triple, a factorization which reduces the size of the stored data. For example, for the triples of Figure 2, the file *exhibited* contains the following lines:

`:guernica :reinasofia .`
`:thethinker :museerodin .`

The above vertical partitioning scheme is used by systems described in [46, 70, 93]. It is reminiscent of the vertical RDF partitioning proposed in [1, 81] for centralized RDF stores.

Following the property-based partitioning scheme, all triples using the special built-in RDF property `:type` are located in the same partition. However, because such triples appear very frequently in RDF datasets, this leads to a single very big file containing such triples. In addition, most triple patterns in SPARQL queries having `:type` as a property usually have the object bound as well. This means that a search for atoms of the form `(:`x`, :type, :t1)`, where `:t1` is a concrete URI, in the partition corresponding to the `:type` property is likely to be inefficient. For this reason, in HadoopRDF [46] the `:type` partition is further split based on the object value of the triples. For example, the triple `(:picasso, :type, :cubist)` is stored in a file named *type#cubist* which holds all the instances of class `:cubist`. This partitioning is performance-wise interesting for the predicate `:type`, because the object of such triples is an RDFS class and the number of distinct classes appearing in the object position of `:type` triples is typically moderate. Clearly, such a partitioning is less interesting for properties having a large number of distinct object values, as this would lead to many small files in the DFS. Since the load at the server in charge of a HDFS cluster (the so-called *namenode*) is proportional to the number of files, a partitioning which leads to many small ones is better avoided.

HadoopRDF [46, 45] goes one step further by splitting triples that have the same property based also on the RDFS class the object belongs to (if such information exists). This can be determined by inspecting all the triples having property `:type`. For example, the triple `(:picasso, :paints, :guernica)` would be stored in a file named $paints\#artifact$ because of the triple `(:guernica, :type, :artifact)`. This grouping helps for query minimization purposes, as we shall see in the next section.

An alternative vertical partitioning model could split triples based on their subject or object value, instead of the property. However, this would lead to a high number of very small files because of the number of distinct subject or object values appearing in RDF datasets, which is to be avoided as explained above. In addition, most SPARQL queries specify the property in the triple patterns, while unspecified subject or object values are more common. Thus, property-based partitioning is generally considered to make selective data access more likely when processing queries, by giving directly access to the triples having a certain property value.

4.1.3 Entity class-based model

A different approach is proposed in EAGRE [92] where the goal is to reduce the I/O cost incurred during query processing, especially for queries with *range* filter expressions (such as $<, >$) and *order* constraints (such as ORDERBY). A distributed I/O scheduling solution is proposed, for finding the data blocks most likely to contain the answers to a query. This scheduling is based on an entity-based compression scheme for RDF. First, the RDF graph is transformed into an entity graph where only nodes that have out-going edges are kept. Each subject of the RDF triples of this graph is called an entity. Entities with similar properties (according to a similarity measure provided by the authors) are grouped together into an entity class. The compressed RDF graph contains only entity classes and the connections between them (properties).

The discovery of the entity classes is performed in two MapReduce jobs. Once they have been found, the entity classes are interlinked locally at each node, based on the connections of the entities they contain, and then, globally at a central node. The global compressed entity graph is then partitioned using METIS [58], a well-known graph partitioning tool. Entities then are placed according to the partition set they belong to.

The storage layout at each node is designed with two purposes: (i) given a query, find the data block of the triples matching the query and (ii) sort triples based on property or object values, so that queries with result order constraints can be efficiently answered. For these reasons, entities in a class having similar property-object values are viewed as high-dimensional data records, and are indexed according to a specialized multidimensional indexing technique [47], based on the so-called *space filling curves*. This is a Hilbert-inspired method of mapping a multidimensional space along fewer or a single dimension, so that data indexing can be performed (and the index exploited) more efficiently.

4.2 Storing and indexing RDF data in key-value stores

Key-value stores provide efficient, fine-grained storage and retrieval of data, well suited to the small granularity of RDF. We classify works in this area into *triple-based* and *graph-based* ones: the former treat RDF data as a collection of triples, while the latter adopt a graph perspective.

4.2.1 Triple-based RDF key-value stores

RDF indexing has been thoroughly studied in a centralized setting [60, 88]. Given the nature of RDF triples, many common RDF indices store in fact all the RDF dataset itself, eliminating the need for a "store" distinct from the index. Thus, *indexing* RDF data in a key-value store coincides with *storing* it there.

Notations. We will use the following notations. To index RDF the values of subjects (S), predicates (P) and objects (O) are used. We may also use the three token strings $\underline{S}$, $\underline{P}$ and $\underline{O}$ to differentiate data that needs to be treated as a subject, predicate, and object, respectively. We use the symbol $\|$ to denote string concatenation and ϵ for the empty string.

We denote *an indexing scheme in a key-value store* by a concatenation of three |-separated symbols ($K|A|V$), specifying (i) which piece of data is the item key (K), (ii) which is the attribute name (A) and finally (iii) which is the attribute value (V). In the case of extended key-value stores, we denote by $\{SA\}$ the superattributes.

In contrast to most centralized RDF stores, which use an extensive indexing scheme, systems relying on key-value stores use only some of the possible indices. For instance, centralized stores such as Hexastore [88] and RDF-3X [60] use all $3! = 6$ permutations of subject, predicate, object to build indices that provide fast data access for all possible triple patterns and for efficiently performing merge-joins.

Table 2: SPO index in a key-value store with a hash index.

item key	(attr. name, attr. value)
:picasso	(:paints, :guernica), (:name, "Pablo"), (:type, :cubist)
:rodin	(:creates, :thethinker), (:name, "Auguste"), (:type, :sculptor)
:guernica	(:exhibited, :reinasofia), (:type, :artifact)
:thethinker	(:exhibited, :museerodin)
:reinasofia	(:located, :madrid)
:museerodin	(:located, :paris)
:Cubist	(:sc, :painter)
:Sculptor	(:sc, :artist)
:Painter	(:sc, :artist)

RDF-3X [60] additionally uses aggregated indices on subsets of the subject, predicate, object, resulting in a total of 15 indices. However, this extensive indexing scheme has a significant storage overhead, which is amplified in a distributed environment (where data is replicated for fault-tolerance). As a consequence, the majority of existing RDF systems built on top of key-value stores use three indices, which turn out to be sufficient for providing efficient access paths to all possible triple patterns. The three permutations massively used by today's systems are: subject-predicate-object (SPO), predicate-object-subject (POS) and object-subject-predicate (OSP). Typical key-value RDF stores materialize each of these permutations in a separate table (or collection).

Depending on the specific capabilities of the underlying key-value store, different designs have been chosen for the key and values. For example, each one of the triple's elements s, p, o can be mapped to the key, attribute name and value of the key-value store, or a concatenation of two or even three of the elements can be used as the key. One of the criteria to decide on the design is whether the key-value store offers a hash-based or sorted index. In the first case, only exact lookups are possible and thus, each of the triples' element should be used as the key. In the latter case, combinations of the triples' element can be used since prefix lookup queries can be answered.

Table 2 shows a possible design for the SPO index in a hash-based key-value store using the RDF running example of Figure 1. Subjects are used as the keys, predicates are used as the attribute names and objects as the attribute values. Similarly, indices POS and OSP are constructed. When the key-value store offers a sorted index on the key, any concatenation of the triples' elements can be used as the key. The order in which the elements are concatenated is determined by the type of index (whether it is SPO, POS or OSP). For instance, an extreme case for the SPO index uses the concatenation of all three triples' elements as the key, while the attribute names and values are empty.

Representative systems using key-value stores as their underlying RDF storage facility include Rya [68] which uses Apache Accumulo [7], CumulusRDF [55] based on Apache Cassandra [8], Stratustore [78] relying on Amazon's Sim-

Table 3: Indices used by RDF systems based on key-value stores.

System	**Key-value store**	**Index type**	**SPO**	**POS**	**OSP**
Rya [68]	Accumulo [7]	Sorted	$SPO\vert\epsilon\vert\epsilon$	$POS\vert\epsilon\vert\epsilon$	$OSP\vert\epsilon\vert\epsilon$
H_2RDF [66]	HBase [10]	Sorted	$SP\vert O\vert\epsilon$	$PO\vert S\vert\epsilon$	$OS\vert P\vert\epsilon$
AMADA [11]	DynamoDB [13]	Hash	$S\vert P\vert O$	$P\vert O\vert S$	$O\vert S\vert P$
MAPSIN [76]	HBase [10]	Sorted	$S\vert P\vert O$	-	$O\vert S\vert P$
Stratustore [78]	SimpleDB [13]	Hash	$S\vert P\vert O$	-	-
CumulusRDF-hierarchical [55]	Cassandra [8]	Hash/Sorted	$S\vert\{P\}O\vert\epsilon$	$P\vert\{O\}S\vert\epsilon$	$O\vert\{S\}P\vert\epsilon$
CumulusRDF-flat [55]	Cassandra [8]	Hash/Sorted	$S\vert PO\vert\epsilon$	$PO\vert S\vert\epsilon$ $PO\vert\underline{P}\vert P$	$O\vert SP\vert\epsilon$

pleDB [13], H_2RDF [66], built on top of HBase [10] and AMADA [11] which uses Amazon's DynamoDB [29].

Table 3 outlines for each system the key-value store used, the type of index data structure provided by the key-value store, and the indices created by the RDF storage platform within the store. Observe that in some cases some data positions within the key-value stores are left empty (ϵ).

An important issue in such approaches is the high skew encountered in the distribution of the property values, i.e., some property values appear very frequently in the RDF datasets. In this case, a storage scheme using the property as the key leads to a table with a few but very large rows. Because in HBase, and other key-value stores, all the data of a row is stored at the same machine, machines corresponding to very popular property values may run out of disk space and further cause bottlenecks when processing queries. For this reason, MAPSIN discards completely the POS index, although techniques for handling such skew, e.g., splitting the long list to another machine [3] are quite well understood by now. To handle property skew CumulusRDF, built on Cassandra, builds a different POS index. The POS index key is made of the property and object, while the subject is used for the attribute name. Further, another attribute, named $\underline{P}$, holds the property value. The secondary index provided by Cassandra, which maps values to keys, is used to retrieve the associated property-object entries for a given property value. This solution prevents overly large property-driven entries, all the while preserving selective access for a given property value.

4.2.2 Graph-oriented storage

A very recent key-value store for RDF is Trinity.RDF [91], proposed by Microsoft. Trinity.RDF is a graph-based RDF engine, which, unlike the systems described above, considers RDF graphs holistically at the level of the storage. The underlying storage and indexing mechanism is Trinity [77], a distributed graph system built on top of an in-memory key-value store. An RDF graph is split in disjoint parts among the cluster machines by hashing the values of the nodes of the RDF graph (subjects and objects of RDF triples). Figure 8 depicts a possible partitioning of the RDF graph of Figure 1 in a cluster of two machines.

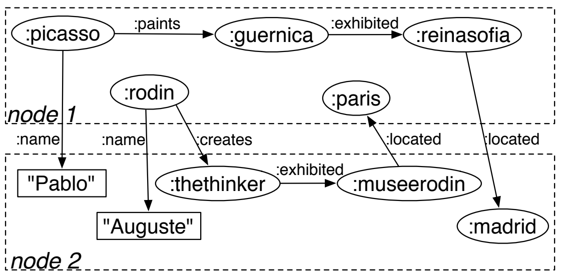


Fig. 8: Trinity.RDF graph partitioning in a 2-machines cluster.

Because some edges inevitably will connect nodes from distinct partitions, each machine will also store some such edges, that are outgoing from the resources assigned to that machines. In other words, these are triples whose subject URI is part of that machine's partition, while the object URI is not.

Within each machine, the URIs (or constants) labeling the RDF graph nodes are used as keys, while two adjacency lists are stored as values: one of the incoming edges and another for the outgoing ones. In our classification, this corresponds to an SPO and an OPS index, respectively. Since these two indices do not allow retrieving triples *by the value of their property*, e.g., to match `(?x, :name, ?y)`, Trinity.RDF comprises a separate index whose keys are the property values, and whose values are two lists, one with the subjects and another with the objects of each property. This approach amounts to storing the indices PS and PO, in our notation.

Table 4 illustrates the RDF graph partitioning of Figure 8 in the key-value store of Trinity. Each node in the graph is stored as a key in one of the two machines together with its incoming and outgoing edges; graph navigation is implemented by lookups on these nodes. The example shown in Table 4 is the simplest way of RDF indexing in Trinity, where IN and OUT denote the lists of incoming and outgoing edges, respectively. In [91], the authors propose a further modification of this storage model aiming at decreasing communication costs during query processing. This is achieved by partitioning the adjacency lists of a graph node

Table 4: Indexing in Trinity.RDF in a 2-machines cluster: key-value store at machine 1 (left) and machine 2 (right).

Key	Value
:picasso	OUT(:paints, :guernica)
	OUT(:name,"Pablo")
:guernica	IN(:paints, :picasso)
	OUT(:exhibited,:reinasofia)
:reinasofia	IN(:exhibited, :guernica)
	OUT(:located, :madrid)
:rodin	OUT(:creates, :thethinker)
	OUT(:name,"Auguste")
:paris	IN(:located,:museerodin)
:paints	{:picasso}, {:guernica}
:name	{:picasso, :rodin}, {-}
:located	{:reinasofia}, {:paris}
:exhibited	{:guernica}, {:reinasofia}
:creates	{:rodin}, {-}

Key	Value
:marid	IN(:located, :reinasofia)
:museerodin	IN(:exhibited, :thethinker)
	OUT(:located, :paris)
:thethinker	IN(:creates,:rodin)
	OUT(:exhibited,:museerodin)
"Pablo"	IN(:name,:picasso)
"Auguste"	IN(:name,:rodin)
:located	{:reinasofia}, {:madrid}
:creates	{-}, {:thethinker}
:exhibited	{:thethinker}, {:museerodin}
:name	{-}, {"Pablo","Auguste"}

by machine so that only one message is sent to each machine regardless of the number of neighbours of that node.

4.3 Federating multiple centralized RDF stores

This category comprises systems that exploit in parallel a set of centralized RDF stores distributed among many nodes.

These systems have a master/slave architecture, where the master is responsible for partitioning and placing the RDF triples in the slave nodes. Each slave node stores and indexes its local RDF triples in a centralized RDF store. The goal is to partition the RDF data in a way that enables high parallelization during query evaluation while striving to minimize communication among the slave nodes. Such approaches include [37, 43, 44].

[44] was the first work to follow this path. RDF graphs are partitioned by METIS [58], which splits the vertices of a graph into k partitions so that a minimum number of edges is cut (an edge cut occurs when the subject and object of a triple are assigned to different partitions). The number of partitions k is the number of available slave nodes. Triples whose predicate value is `:type` are removed from the partitioning process as they reduce the quality of the graph partitioning.

Assuming that we have four nodes available, Figure 9(a) shows a possible output of METIS for the RDF graph of Figure 1. Placement is done by assigning the triple to the partition to which its subject belongs (referred as *directed 1-hop guarantee*) or by assigning the triple to the partitions of both its subject and the object (referred as *undirected 1-hop guarantee*). To further reduce communication network during query evaluation, the authors of [44] allow for even more replication of those triples that are at partition boundaries. A *directed (undirected) n-hop guarantee* can be obtained through replication; it ensures that any triples forming a directed (undirected) path of length n will be located within the same partition. Figures 9(b) and 9(c) depict the placement of triples to the four partitions with undirected 1-hop and 2-hop guarantee, respectively. The triple placement algorithm is performed with Hadoop jobs, while the local triples at each node are stored in RDF-3X [60].

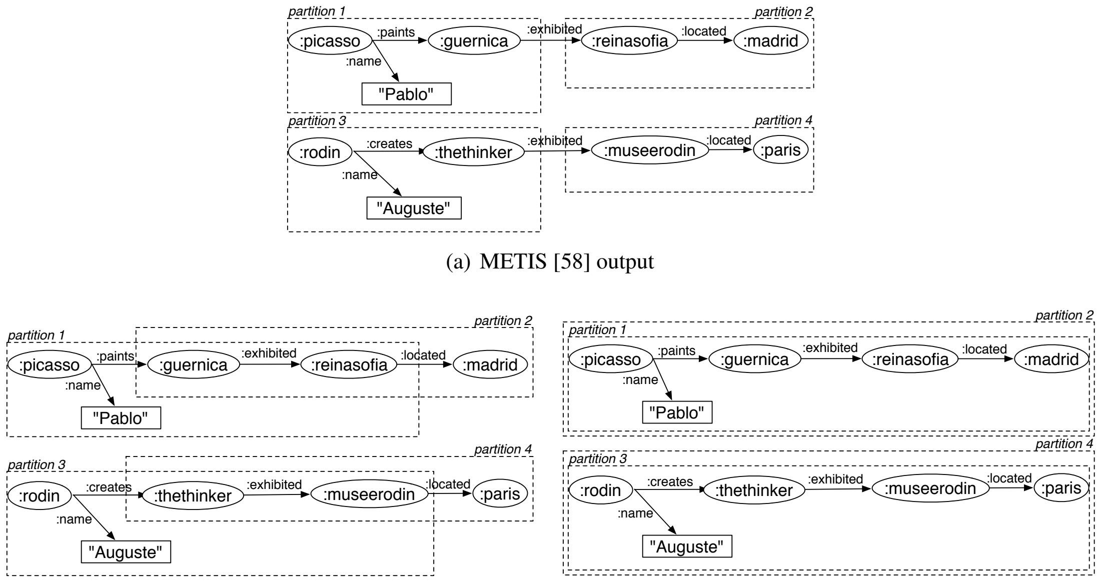
WARP [43] extends the partitioning and replication techniques of [44] to take into account a query workload in order to choose the parts of RDF data that need to be replicated. Thus, rarely-used RDF data does not need to be replicated, leading to a reduced storage overhead compared to the one of [44].

Partout [37] is a distributed RDF engine also concerned with partitioning RDF, inspired by a query workload, so that queries are processed over a minimum number of nodes. The partitioning process is split in two tasks: fragmentation, i.e., split the triples into pieces, and fragment allocation, i.e., in which node each piece will be stored. The fragmentation is based on a horizontal partitioning of the triple relation based on the set of constants appearing in the queries while the fragment allocation is done so that most of the queries can be executed locally at one host but at the same time maintaining load balancing. This is done by counting for each fragment the queries that need to access it, as well as the triples matching the fragment. The triple partitioning process takes into consideration load imbalances and space constraints to assign the partitions to the available nodes.

Finally, in [90], HadoopDB [4] is used, where RDF triples are stored in a DBMS at each node. Triples are stored using the vertical partitioning proposed in [1], i.e., one table with two columns per property. Triple placement is decided by hashing the subjects and/or objects of the triples according to how the properties are connected in the RDF Schema. If the RDF Schema is not available, a schema is built by analyzing the data at loading time.

4.4 Hybrid approaches

In the third category, we classify systems which exploit a combination of a distributed file system, a key-value store and a centralized RDF store. Such a system is AMADA [11,



(a) METIS [58] output

(b) 1-hop undirected guarantee

(c) 2-hop undirected guarantee

Fig. 9: Partitioning and placement of [44].

22, 21], which is built upon services provided by commercial clouds. In AMADA, RDF data is stored in a distributed file system, and the focus is on efficiently routing queries to only those datasets that are likely to have matches for the query. Selective query routing reduces the total work associated to processing a query; in a cloud environment, this also translates in financial cost savings. Thus, AMADA takes advantage of: large-scale stores for the data itself; the fine-grained search capabilities of a fast key-value store for efficient query routing and an out of the box centralized RDF store for caching relative datasets and efficiently answer queries.

AMADA stores RDF datasets within Amazon's Simple Storage Service (S3), a distributed file system for raw (file) storage. Then, indices are built in Amazon's key-value store, DynamoDB [33]. These indices differ from the ones of Section 4.2 because they are used for mapping RDF elements to the RDF datasets containing them and not for indexing RDF triples themselves. For example, following the notation introduced in Section 4.2, the index S|D|ϵ shows which subject values S can be found in which RDF datasets D. Similar indices are used for property and object values or a combination of them. In [21] the following indexing schemes are proposed and experimentally compared:

– Term-based indexing which uses the index T|D|ϵ, where T is any RDF term regardless of its position.
– Attribute-based indexing which uses three indices: S|D|ϵ, P|D|ϵ, O|D|ϵ.
– Attribute-subset indexing which uses seven indices: $S|D|\epsilon$, $P|U|\epsilon$, $O|U|\epsilon$, $SP|U|\epsilon$, $PO|U|\epsilon$, $SO|U|\epsilon$, $SPO|U|\epsilon$.

The indices outlined above enable efficient query processing in AMADA through routing incoming queries (only) to those the dataset(s) that may have useful results, as we explain in the next section. These datasets are then loaded in a centralized RDF store.

4.5 Summary

Table 5 summarizes the back-end and storage layout schemes used by the RDF stores. For each work, we also present the benefit of its chosen storage scheme, outlining three classes of back-ends used for storing RDF triples: distributed file systems, key-value stores and centralized RDF stores.

We observe that almost all systems based on a key-value store adopt a 3-index scheme, more precisely: SPO, POS, and OSP. Trinity.RDF [91] is the exception, with its second-level index as described above. Finally, the only works that consider a query workload for the partitioning process are WARP [43] and Partout [37].

5 Cloud-based SPARQL query processing

A second important angle of analysis of cloud-based RDF platforms comes from their strategy for processing SPARQL queries. From this perspective, we identify the following main classes:

– systems following a relational-style query processing strategy; and

Table 5: Comparison of storage schemes.

System	Storage back-end	Storage layout/partitioning	Benefit
SHARD [72]	DFS	Triple-based files (1 line for each subject)	Simplify storage
PigSPARQL [75]	DFS	Triple-based files (1 line for each triple)	Simplify storage
HadoopRDF [46, 45]	DFS	Property-based files	Reduce I/O and data processing
RAPID+ [70, 51]	DFS	Property-based files	Reduce I/O and data processing
Zhang et al. [93]	DFS	Property-based files	Reduce I/O and data processing
EAGRE [92]	DFS	Graph partitioning (METIS) on compressed entity graph	Reduce I/O and data processing
H_2RDF [66]	Key-value store	3 indices (SPO, POS, OSP)	Fast data access
Rya [68]	Key-value store	3 indices (SPO, POS, OSP)	Fast data access
AMADA [21, 11, 22]	Key-value store	3 indices (SPO, POS, OSP)	Fast data access
MAPSIN [76]	Key-value store	3 indices (SPO, POS, OSP)	Fast data access
Stratustore [78]	Key-value store	1 index (SPO)	Fast data access
CumulusRDF [55]	Key-value store	3 indices (SPO, POS, OSP)	Fast data access
Trinity.RDF [91]	Key-value store	Graph-based indexing (SPO, OPS, PS, PO)	Fast data access
Huang et al. [44]	Centralized RDF store	Graph partitioning by METIS	Reduce communication cost
WARP [43]	Centralized RDF store	Graph partitioning based on query workload	Reduce communication cost
Partout [37]	Centralized RDF store	Horizontal fragmentation based on query workload	Reduce communication cost
Wu et al. [90]	Centralized RDBMS	Hash-based horizontal partitioning and property-based relations on each node	Reduce communication cost

– systems using graph exploration techniques based on the graph structure of the data.

While there is only one work that uses graph techniques, many works have relied on relational processing strategies. We present them below.

5.1 Relational-style SPARQL query processing in the cloud

Works in this category can be classified according to the taxonomy of Figure 10. From a database perspective, two are the basic operations in evaluating a SPARQL query: (i) data access paths and (ii) join evaluation. The first operation consists of retrieving from the storage data matching some fragment of the query (for instance, one or several triple patterns), while the latter determines how these pieces are joined to form the final answer.

The platforms on which the systems we survey are built provide a big opportunity for parallelizing query processing and thus, for achieving better query performance. For this reason query decomposition techniques need to be revised to tackle this challenge. In addition, for efficiency reasons, query processing in distributed architectures should reduce the shuffling of data between nodes. We provide more details on these aspects in Section 5.1.2.

5.1.1 Data access paths

The data access paths available to the query engine are directly determined by the underlying storage facility. We categorize them accordingly.

Distributed file systems. When RDF data is stored in a distributed file system, one can only scan files for the triples that match a given triple pattern.

In systems that store RDF data according to the triple model described in Section 4.1.1, all the files are scanned and a selection operation is performed to match the triple pattern. In [71, 75] the selection is performed in the map phase of the corresponding MapReduce job.

In systems based on the vertical partitioning (Section 4.1.2), triple pattern matching is performed by selecting the files named after the property of the triple pattern. If the subject and/or object of the triple pattern is also constant (specified by the query), the corresponding selection conditions are enforced on the data retrieved from those files. In [46, 70, 93] the selection is performed in the corresponding map phase. As in the case of centralized stores, this kind of partitioning works well in the cases of triple patterns with a bound predicate. Such triple atoms have been reported to be frequent in real-world SPARQL queries, amounting to about 78% of the DBPedia query log and 99.5% of the Semantic Web Dog query log according to [12]. However, when the property attribute of a triple pattern is unbound, all files residing in the DFS need to be scanned.

An interesting aspect of HadoopRDF [46] data access can be seen as a form of query minimization during the file selection process. For a triple pattern $t_1 = (s_1, p_1, o_1)$ where p_1 is distinct from `:type` and o_1 a variable, HadoopRDF checks if the type of the object is specified by another triple pattern $t_2 = (o_1, \texttt{:type}, o_2)$ in the query. If this is the case, the selected file is only $p_1\#o_2$, and t_2 is removed from the query. Otherwise, all files prefixed with p_1 are selected.

In EAGRE [92] a distributed I/O scheduling solution is proposed for reducing the I/O cost incurred before the filtering of the map phase, especially for queries with range and order constraints. Query evaluation is postponed until the I/O scheduling has determined which are the data blocks

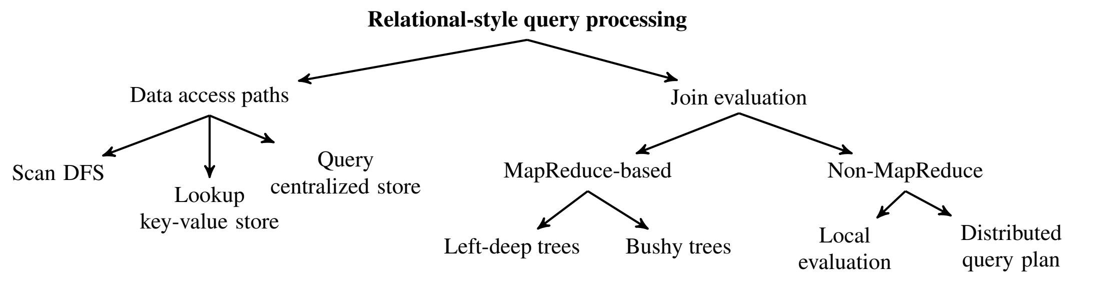


Fig. 10: Taxonomy of relational-style query processing strategies.

that contain answers to a given query (and thus, need to be scanned).

Key value stores. When RDF triples are stored in a key-value store, triple pattern matching becomes more efficient due to the indexing capabilities.

Access paths depend on the way RDF data is indexed in the key-value store and the kind of index the underlying key-value store supports (hash- or sorted index). If it is a hash index, a direct lookup is necessary to retrieve the values of a given key; in contrast, if it is a sorted index, a prefix lookup is also possible. In some cases, post-processing is required after the lookup to further filter out triples which do not match the triple pattern.

Table 6 shows the access paths provided by each such system, for each possible query triple pattern template. We denote by DL(T) a direct lookup to a table T, by PL(T) a prefix lookup and by S(T) a scan over the whole table T. In addition, we use $csel$ to denote that an extra selection operation is required at the client side, while $ssel$ specifies when the selection can be performed on the server side (where the key-value store is located). In the case of Stratustore where SimpleDB is used, there is also the possibility of `SELECT` queries which return keys and values by specifying the columns. We denote such cases by SDB(T).

For a boolean triple pattern (all its elements are constants) a direct lookup to any of the available tables can be performed and then (depending on the indexing strategy) a selection may be needed. Similarly, for a variable-only triple pattern, a scan over any of the available tables is suitable. For the remaining possible combinations of variables, constants and URIs in triple patterns, we may need to perform either a direct lookup, or a direct lookup with some post-filtering, or a prefix lookup.

Federated centralized RDF stores. Huang et al. [44] and WARP [43] aim at pushing as much as possible query processing to the centralized RDF stores available on each node. After the SPARQL query has been decomposed into subqueries, data access is implemented by sending each subquery to all instances of the RDF store in parallel. This is necessary since there is no index to map the data to the partitions it belongs. For instance, if the undirected 2-hop guarantee of [44] is provided by the store (Figure 9(c)), the query of Figure 4 can be evaluated by sending it to all four partitions. If a query cannot be completely answered from the underlying store, further joins are required to combine the subquery results, as we explain in the next section.

While [43, 44] lack indices for routing the subqueries to the partitions containing results, Partout [37] identifies the nodes that are relevant to a specific triple pattern at the coordinator node. It builds mappings of terms appearing in the query workload to nodes during the partitioning, and exploits it afterward for query processing.

All these three works use RDF-3X [60] as the underlying storage for each partition. As a consequence, local data access is efficiently supported for each triple pattern in the query, based on the local indices built by RDF-3X on all (combinations of) triple elements.

Finally, in [90] SPARQL queries are also decomposed into subqueries, based on the data partitioning. The subqueries are translated into SQL queries and sent for evaluation to the underlying RDBMS of HadoopDB [4].

Hybrid approaches. AMADA [11, 21, 22] uses a hybrid approach where data resides in files in a cloud-store, indices pointing to the files are kept in a key-value store, while a centralized RDF store loads and evaluates the query at runtime. Query processing is achieved by identifying a (hopefully tight) superset of the RDF datasets which contain answers to a given query, based on the available indices (see Section 4.4). Then, the selected RDF datasets are loaded at query time in a centralized state-of-the-art RDF store which gives the answer to the query.

5.1.2 Join evaluation

A first aspect we consider is the method used to implement (potentially distributed) joins. These are natively not supported in key-value stores, nor in the MapReduce framework. For this reason, very early works on indexing RDF in key-value stores do not handle joins; for instance, Cumulus-RDF [55] supports only single triple pattern queries. Among

Table 6: Triple pattern access paths in key-value stores.

Triple pattern	**Rya [68]**	**H^{2}RDF [66]**	**AMADA [11]**	**MAPSIN [76]**	**Stratustore [78]**	**CumulusRDF hierarchical** [55]	**CumulusRDF flat** [55]
(s, p, o)	DL(*)	DL(*)+$csel$	DL(*)+$csel$	DL(*)+$ssel$	DL(*)+$csel$	DL(*)+$csel$	DL(*)+$csel$
$(s, p, ?o)$	PL(SPO)	DL(SP|O)	DL(S|P|O)+$csel$	DL(S|P|O)+$ssel$	DL(S|P|O)+$csel$	2× DL(S|{P}O|ϵ)	DL+PL(S|PO|ϵ)
$(s, ?p, o)$	PL(OSP)	DL(OS|P)	DL(O|S|P)+$csel$	DL(O|S|P)+$ssel$	DL(S|P|O)+$csel$	2× DL(O|{S}P|ϵ)	DL+PL(O|SP|ϵ)
$(s, ?p, ?o)$	PL(SPO)	PL(SP|O)	DL(S|P|O)	DL(S|P|O)	DL(S|P|O)	DL(S|{P}O|ϵ)	DL(S|PO|ϵ)
$(?s, p, o)$	PL(POS)	DL(PO|S)	DL(P|O|S)+$csel$	DL(O|S|P)+$ssel$	SDB(S|P|O)	DL(P|{O}S|ϵ)	DL(PO|S|ϵ)
$(?s, p, ?o)$	PL(POS)	PL(PO|S)	DL(P|O|S)	S(*)+$ssel$	SDB(S|P|O)	DL(P|{O}S|ϵ)	DL(PO|$\underline{\text{P}}$|P)+ DL(PO|S|ϵ)
$(?s, ?p, o)$	PL(OSP)	PL(OS|P)	DL(O|S|P)	DL(O|S|P)	S(*)+$csel$	DL(O|{S}P|ϵ)	DL(O|SP|ϵ)
$(?s, ?p, ?o)$	S(*)	S(*)	S(*)	S(*)	S(*)	S(*)	S(*)

the other systems, two join evaluation methods prevail: either by using the MapReduce framework or by performing the join out of MapReduce, often at a single site.

Another important aspect of parallel SPARQL join evaluation is the way a query is decomposed into subqueries to be processed in parallel; this query decomposition stage is often dictated by the data partitioning method. The query decomposition can lead to either left-deep or bushy query plans being built. The search space for finding the optimal left-deep query plan is of size $n!$ for a query with n triple patterns. However, in a distributed/parallel setting, this approach only exploits intra-operator parallelism (evaluating one operator in parallel), but lacks inter-operator parallelism (evaluating multiple operators in parallel). Bushy trees are better suited for parallel query processing since multiple operators can be evaluated in parallel. However, the search space of bushy query plans is exponential in the size of the query, $O(n \times 2^n)$ for star queries and $O(3^n)$ for path queries (where n is the number of relations) [62]. Enumerating and estimating the cost of such large plan sets can be inefficient for large queries. For this reason, many of the works resort to simple left-deep plans or use a hybrid of the two: plans that are bushy only at the leaves level, while the intermediate relations are joined sequentially in a left-deep manner. We term the latter *leaf-bushy* plans, and reserve the standard *bushy* term for those that are bushy at all levels.

MapReduce-based joins. The first system to use MapReduce for SPARQL query evaluation is SHARD [71]. In this system, one MapReduce job is created for each triple pattern, and an extra (last) job is created for removing redundant results (if necessary) and projecting the corresponding values. In the map phase of each job, the triples matching the triple pattern are identified and sent to the reducers. In the reduce phase, the matched triples are joined with the intermediate results of the previous triple patterns (if any). Conceptually, SHARD's query evaluation strategy leads to left-deep query plans, which correspond to a sequence of MapReduce jobs and to potentially long query evaluation time.

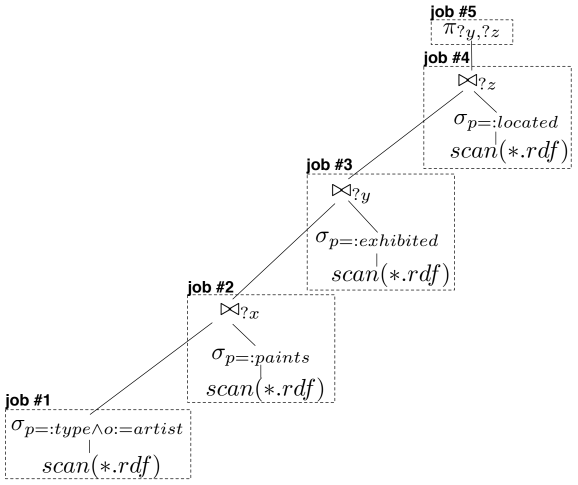


Fig. 11: Left-deep query plan in MapReduce as produced by SHARD [71].

Figure 11 shows how a left-deep query plan of the example query of Figure 4 can be evaluated in MapReduce. Note that for a 4-triple pattern query five jobs are required, while all data is scanned 4 times.

In [75] the authors propose a mapping from full SPARQL 1.0 to Pig Latin [61], a higher-level close to the nested relational algebra, providing primitives such as filter, join and union; PigLatin is then compiled into MapReduce. Each triple pattern is transformed into a Pig Latin filter operation, following which the corresponding data sets are joined in a sequence, also corresponding to a left-deep query plan. Left-outer joins and unions are also used for more complex SPARQL queries, featuring the OPTIONAL and UNION clauses. Standard optimization techniques like pushing projections and selectivity-based join reordering are also used. Query evaluation in [75] also leads to left-deep query plans.

In contrast with the above, the parallelization possibilities offered by MapReduce make evaluation through bushy plans attractive, since many operators can be evaluated in parallel. Given that the full search space of bushy plans is large, heuristics are often used to identify a plan close to the

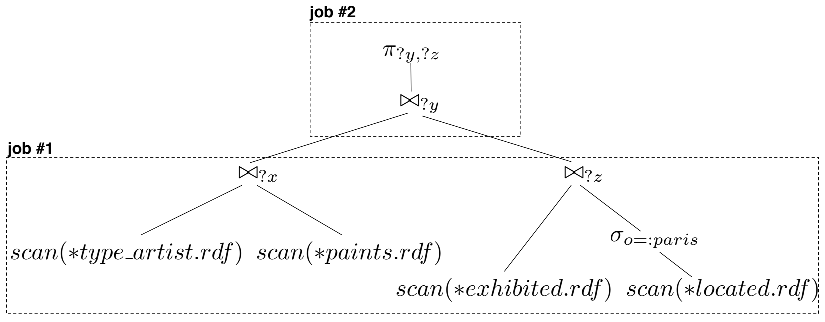


Fig. 12: Bushy query plan in MapReduce as produced by HadoopRDF [46].

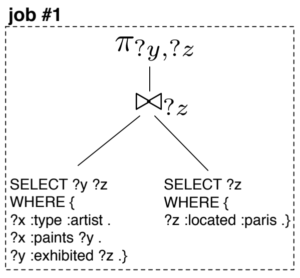
optimal. The heuristic used mostly in recent MapReduce-based proposals is to produce a query plan that requires *the least number of jobs* since the overhead of initializing a MapReduce job is significant [27]. Although traditional selectivity-based optimization techniques for finding optimal query plans may decrease the intermediary results, they may also lead to a growth in the number of jobs and thus, to worse query plans with respect to query response time. Therefore, the ultimate goal of such proposals is to produce query plans in the shape of balanced bushy trees with the minimum possible height.

HadoopRDF [46], H_2RDF [66] and RAPID+ [51, 70] are systems that try to achieve the above goal. A join among two or more triple patterns is performed on the same single variable. Within one job, one or more joins can be performed as long as they are on different variables. When the query has only one join variable, only one job suffices for query evaluation. However, as this is not always the case, query planning is required to output a sequence of MapReduce jobs.

In HadoopRDF [45, 46] a heuristic is used to bundle as many joins as possible in each job, leading to query plans with few MapReduce jobs. The same heuristic is used in H_2RDF [66] for non-selective queries. Although H_2RDF stores the RDF data in a key-value store, it uses MapReduce query plans to evaluate queries whose results are estimated to be large and thus, benefit from parallelization.

Figure 12 demonstrates a possible query plan produced by HadoopRDF for the example query of Figure 4. In the first job, the joins on variables $?x$ and $?z$ are computed between the first and the last two triple patterns, respectively. The second job joins the intermediate results of the first job on variable $?y$.

In RAPID+ [51, 70] an intermediate nested algebra is proposed for increasing the degree of parallelism when evaluating joins and thus reducing the number of MapReduce jobs. This is achieved by treating star joins (groups of triple patterns having as subject the same variable) as groups of triples and defining new operators on these triple groups. Queries with k star-shaped subqueries are translated into



```
SELECT ?y ?z
WHERE {
?x :type :artist .
?x :paints ?y .
?y :exhibited ?z .}
```

```
SELECT ?z
WHERE {
?z :located :paris .}
```

Fig. 13: Query plan as executed in Huang et al. [44].

a MapReduce program with k MapReduce jobs: 1 job for evaluating all star-join subqueries and $k-1$ jobs for joining the subquery results thus obtained. Conceptually, such query plans are leaf-level bushy plans since the intermediary results of the star-join subqueries are evaluated pair-wise and sequentially. The proposed algebra is integrated with Pig Latin [61]. In [52], the authors extend RAPID+ with a scan sharing technique applied in the reduce phase to optimize queries where some property value appears more than once.

In [76] a Map-side index nested loops algorithm is proposed for joining triple patterns. The join between two triple patterns is computed in the Map-phase of a job by retrieving from the key-value store values that match the first triple pattern, and replacing the values found for the join variable in the second triple pattern. For each value, the 2nd triple pattern is rewritten and the corresponding lookup is performed in the key-value store. No shuffle or reduce phases are required. As an optimization, the triple patterns that share a common variable on the subject (or object) are grouped and evaluated at once in a single Map phase.

In Huang et al. [44] complex queries, which cannot be evaluated completely by the underlying RDF store due to the partitioning scheme, are evaluated within MapReduce for joining the results from different partitions. A query is decomposed in subqueries that can be evaluated independently at every partition and then the intermediary results are joined through MapReduce jobs sequentially. This creates leaf-level bushy trees, and leads to as many MapReduce jobs as there are subqueries. Figure 13 shows how the example query of Figure 4 is evaluated based on a partitioned store providing the 1-hop guarantee shown in Figure 9(b). The query is decomposed into two subqueries; the first one contains three triple patterns, while the second one contains only the last triple pattern. The results from the two subqueries are joined in a MapReduce job. A similar approach is followed in [90], where subqueries are processed sequentially (one job per subquery) and then are joined in a left-deep tree manner (one job per join).

In [93] the authors propose a method for constructing a tree which encapsulates all possible join plans that need to be examined. Then, based on a cost model, the join plan with the minimum cost is chosen by a top-down traversal of the tree and by pruning some of the non-optimal query plans.

In all the above works except [93], joins are evaluated using the standard repartition join algorithm. In [93] either the repartition or the broadcast join is used, depending on whether the property file is small enough to fit into memory. Finally, in [93] and [90] some pruning techniques are used to reduce the number of intermediary results that are shuffled in the network. In the former, the authors use Bloom filters on the subjects or objects of small property files and in the latter the minimum and maximum values of each variable in a job are stored to be used in the subsequent jobs as a filter condition in the SQL query.

Join evaluation outside MapReduce. Systems that execute joins outside MapReduce store their data (i) in key-value stores, or (ii) in centralized RDF stores over multiple nodes.

The former typically implement their own join operators, since key-value stores do not allow for operations across tables (e.g., , joins). Often the evaluation of SPARQL queries in such systems is done *locally* at a single site, outside of the key-value store. For each triple pattern, the appropriate lookups are performed within the key-value store, then the results are joined to produce the final answer. Conceptually, such approaches lead to left-deep plans since no inter-operator parallelism is used.

Rya [68] implements an index nested loops algorithm using multiple lookups in the key value store, similar to [76] but without using MapReduce. For the first triple pattern t_1, a lookup is performed to find bindings for its variables. Then, for the remaining triple patterns t_i, the k values from the triple pattern t_{i-1} are used to rewrite t_i. Then k lookups are performed in the key-value store for t_i. This procedure is performed locally at the server. In AMADA [11], a query with n triple patterns entails n lookups, whose results are joined locally using an in-memory hash join.

An interesting case is H_2RDF [66], which uses MapReduce only for the non-selective queries. For selective queries a centralized index nested loops algorithm is used, similarly with Rya. Stratustore [78] poses a SimpleDB query for each star-join, specifying in the WHERE clause if the attribute name (property) should be equal to a value (object), collects the keys (subjects) and attribute values (objects) specified in the SELECT clause and performs the appropriate joins among them locally.

Distributed engines using centralized RDF stores over multiple nodes also implement their own join operators for joining the intermediary results of subqueries evaluated by the RDF store, when necessary. For instance, in [43], for queries that cannot be evaluated completely independently at each partition, the intermediary results from each partition are gathered in the coordinator node where the joins are performed in a left-deep plan.

In Partout [37] the coordinator is responsible for generating a query plan whose leaves are index scans of triple patterns at the nodes containing relevant data. Because data is stored in RDF-3X, results of the index scans are ordered and thus, the inner operations are merge joins whenever both inputs are ordered on the join attribute. If this is not the case, the inputs need to be sorted or a hash join is used instead. A join operator is executed at one of the nodes of its children, following the optimizer's decision.

5.2 Graph-based SPARQL query processing

Trinity.RDF [91] is based on a graph-structured store, and thus it uses graph exploration instead of relational-style joins, in order to evaluate SPARQL queries.

Data access paths. To match triple patterns, graph exploration needs a constant subject or object (source value) to start with. The node that contains the source value can easily be found through hashing. Then, this node retrieves the nodes responsible of storing the target (object, or subject) values, and sends these nodes a message.

For example, assume the graph partitioning shown in Figure 8 and the triple pattern `(:picasso, ?p, ?o)`. A message is first sent to node 1 to determine the nodes holding the object values `:picasso`; in our example, these are nodes 1 and 2. From node 1 we retrieve the object value `:guernica`, while a message is sent to node 2 where the object value `"Pablo"` is found.

If neither of the subject, object of the triple pattern are constants, the POS index is used to find matches for the given property. Graph exploration starts in parallel from each subject (or object) found and by filtering out the target values not matching the property value. If the predicate is also unbound, then every predicate is retrieved from the POS index.

For example, assume again Figure 8 and the triple pattern `(?s, :name, ?o)`. From the POS index (see Table 4)

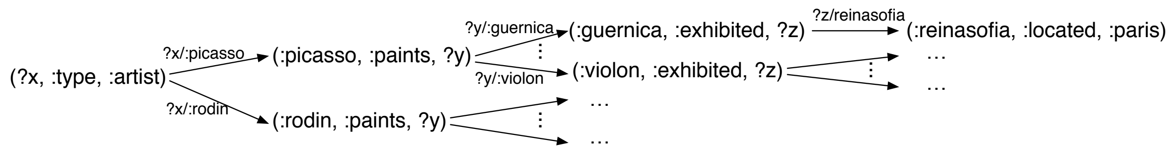


Fig. 14: Graph exploration in Trinity.RDF for query in Figure 4.

two subject values are found: :`picasso` and :`rodin`. For both of these values, the above procedure is again followed; moreover, the object values :`guernica` and :`thinker` are filtered out because the property value is not matched.

Join evaluation. For conjunctive queries, triple patterns are processed sequentially through graph exploration. One triple pattern is picked as the root and a chain of triple patterns is formed. The results of each triple pattern guide the exploration of the graph for the next one. This completely avoids manipulating triple patterns that may match a triple but do not match another one considered previously during query evaluation. Figure 14 illustrates how the query of Figure 4 is processed in Trinity.RDF, assuming the given triple pattern evaluation order.

This graph exploration resembles the nested index loops algorithm also used in Rya [68] and H_2RDF [66] with the difference that only the matches of the immediate neighbors of a triple pattern are kept and not all the history of the triple patterns' matches during the graph exploration and thus, invalid results may be included in the results. For this reason, a final join is required at the end of the process to remove any invalid results that have not been pruned through the graph exploration. This join typically involves a negligible overhead.

Obviously, the order in which the triple patterns are evaluated significantly impacts performance. In [91] the authors propose a cost-based optimization based on dynamic programming and a selectivity estimation technique capturing the correlation between pairs of triples, in order to select a favorable triple pattern evaluation order.

5.3 Summary

Table 7 summarizes the query processing strategy of each system. It shows the correlation of the data access paths with the different join evaluation frameworks used. In addition, it outlines the type of query plans conceptually created and the type of join used by each system. In contrast with our storage-based categorization, where each system fit in only one category (Section 4.5), the query-based classification is less clear-cut, with some systems pertaining to more than one class. For example, H_2RDF uses both a MapReduce-based query evaluation and a local evaluation depending on the query selectivity, while Huang et al. [44] use a hybrid approach between MapReduce and evaluation on multiple centralized RDF stores. We view this diversity as proof of the current interest in exploring various methods – and their combinations – for massively distributed RDF query processing.

6 RDFS reasoning in the cloud

In Section 2 we introduced the role of *inference* (or reasoning) and the important place of *entailed triples* in an RDF data management context. Generally, there are three methods to handle RDFS reasoning:

– *closure computation*: compute and materialize all entailed triples;
– *query reformulation*: reformulate a given query to take into account entailed triples;
– *hybrid*: some mix of the two above approaches.

The first method requires computing the closure prior to query processing, while the second (reformulation) is executed at query time. Finally, in the hybrid approach some entailed data is computed statically and some reformulation is done at query time. A comparison of these RDF reasoning methods can be found for a centralized setting in [38] and for a distributed one in [48].

We classify the cloud-based systems that support RDFS reasoning according to these three categories. We also consider parallel/distributed approaches that were not necessarily intended for the cloud but can be easily deployed therein. The main challenge faced by these systems is to be complete (answer queries by taking into account all the implicit triples), even though the data is distributed. At the same time, the total volume of shuffled data should be minimized, not to degrade performance.

6.1 RDFS closure computation in the cloud

One of the most widely spread inference methods for RDF is the precomputation and materialization of all entailed triples, also called *RDFS closure* computation. This method works in a bottom-up fashion; new RDF triples are exhaustively generated based on the RDFS entailment rules and stored

Table 7: Correlation of data access paths with join evaluation strategies.
(*DFS*: Distributed file system, *KV*: Key-value store, *MR*: MapReduce, *L*: Left-deep plans, *B*: Bushy plans, *LB*: Leaf-level bushy plans, *SRJ*: Standard repartition join, *BJ*: Broadcast join

System	**Data access**	**Join framework**	**Plans**	**Join method**	**Comment**
SHARD [72]	DFS scan (all data)	MR	L	SRJ	
PigSPARQL [75]	DFS scan (all data)	MR	L	SRJ	Translation of SPARQL queries to PigLatin
HadoopRDF [46, 45]	DFS scan (property files)	MR	B	SRJ	
RAPID+ [70, 51, 52]	DFS scan (property files)	MR	LB	SRJ	Scan sharing for queries with repeated properties
EAGRE [92]	DFS scan (relevant blocks)	MR	N/A	N/A	
Zhang et al. [93]	DFS scan (property files)	MR	B	SRJ/BJ	Bloom filters for pruning intermediary results
MAPSIN [76]	KV lookup	MR (map-only)	L	Index nested loops	
H$_2$RDF [66]	KV lookup	Locally/MR	L/B	Index nested loops/SRJ	Evaluation strategy depends on query selectivity
Rya [68]	KV lookup	Locally	L	Index nested loops	
AMADA [21, 11, 22]	KV lookup	Locally	L	Hash join	
Stratustore [78]	KV lookup	Locally	L	N/A	
CumulusRDF [55]	KV lookup	-	-	-	
Huang et al. [44]	SPARQL subqueries to central RDF store	Locally/MR	LB	Central store/SRJ	Depends on data replication
WARP [43]	SPARQL subqueries to central RDF store	Locally/MR	LB	Central store/SRJ	Depends on data replication
Partout [37]	SPARQL subqueries to central RDF store	Distributed	LB	Merge/hash join	
Wu et al. [90]	SQL subqueries to RDBMS	MR	LB	SRJ	Min-max values for pruning intermediary results
Trinity.RDF [91]	KV store lookup	Trinity	L	Traversal of RDF graph	

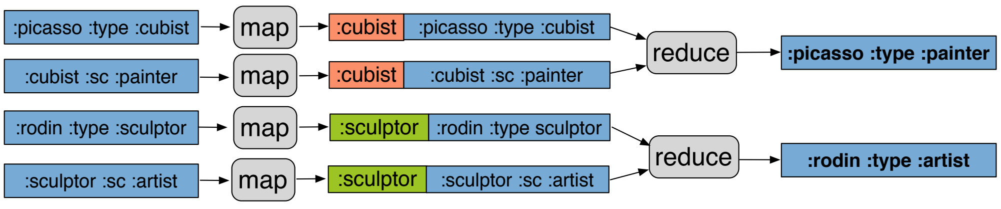


Fig. 15: MapReduce-based application of rule i_2 of Table 1.

until no more new triples can be produced. A query is then evaluated on the RDFS closure and yields a complete answer taking into account both the given RDF triples and the entailed ones. Although this approach has minimal requirements during query answering, it needs a significant amount of time and space to compute and store all inferred data. For this reason, the parallel processing paradigm of MapReduce is suitable for computing the RDFS closure.

One of the first works providing RDFS closure computation algorithms in a parallel environment is WebPie [84]. RDF data is stored in a distributed file system and the RDFS entailment rules of Table 1 are used for precomputing the RDFS closure through MapReduce jobs.

First, observe in Table 1 that the rules having two triples in the body, imply a join between the two triples because they have a common value. See, for example, rule s_1 where the object of the first triple should be the same as the subject of the second one. By selecting the appropriate triple attributes as the output key of the map task, the triples having a common element will meet at the same reducer. Then, at the reducer the rule can be applied to generate a new triple, thus allowing to parallelize inference. Figure 15 illustrates the application of rule i_2 from Table 1 within a MapReduce job. In the map phase, the triples are read and a key-value pair is output. The key is the subject or object of the triple, depending on its type, and the value is the triple itself. All the triples generated with the same key meet at the same reducer where the new triple is produced.

Second, entailed triples can also be used as input in the rules. For instance, in the example of Figure 1 the entailed triple `(:picasso, :type, :painter)` can be used to infer the triple `(:picasso, :type, :artist)`. Thus, to compute the RDFS closure, repeated execution of MapReduce jobs is needed until a fixpoint is reached, that is, no new triples are generated.

In WebPie [84] three optimization techniques are proposed to achieve a fixpoint as soon as possible. The first one starts by the observation that in each RDFS rule with a two-triples body, one of the two is always a schema triple. Since RDF schemas are usually much smaller than RDF datasets, the authors of [84] propose to replicate the schema at each node and keep it in memory. Then, each rule can be applied directly either in the map phase or in the reduce phase of a job, given that the schema is available at each node.

The second optimization consists of applying rules in the reduce phase to take advantage of triple grouping and thus avoid redundant data generation. If the rules are applied in the map phase, many redundant triples can be generated. For example, Figure 16(a) shows that for the application of rule i_3 in the map phase the same triple is produced three times. On the other hand, Figure 16(b) demonstrates how rule i_3 can be applied in the reduce phase causing no redundancy.

Finally, in [84] the authors propose an application order for RDFS rules based on their interdependencies so that the required number of MapReduce cycles is minimized. For example, rule i_3 depends on rule i_1; output triples of i_1 are input triples of i_3. Thus, it is more efficient to apply first rule i_1 and then i_3. Thus, the authors show that one can *process each rule only once* and obtain the RDFS closure with the minimum number of MapReduce jobs.

At the same time as [84], the authors of [87] present a similar method for computing the RDFS closure based on the complete set of entailment rules of [42] in a parallel way using MPI. In [87] they show that the full set of RDFS rules of [42] has certain properties that allow for an *embarrassingly parallel* algorithm, meaning that the interdependencies between the rules can easily be handled by ordering them appropriately. This means that the RDFS reasoning task can be divided into completely independent tasks that can be executed in parallel by separate processes. Similarly with [84], each process has access to all schema triples, while data triples are split equally among the processes, and reasoning takes place in parallel.

Finally, the authors of [84] have extended WebPie in [83] to enable the closure computation based on the OWL Horst rules [80].

6.2 Query reformulation

An alternative technique that has also been adopted for RDFS reasoning is computing only the inferred information that is related to a given query at query evaluation time. This involves *query reformulation* based on the RDF schema and a set of RDFS entailment rules. Thus, reformulation (also) enables query answers to reflect both the given and the entailed RDF triples.

Query reformulation works by rewriting each triple pattern based on the RDF schema and the RDFS entailment rules. This results in a union of triple patterns for each triple pattern of the initial query. For instance, the single triple pattern query $q =$`(?x, :type, artist)` should be rewritten according to the RDF schema of Figure 1 and rule i_2 as q':

$q' =$ `(?x, :type, :artist)`$\vee$`(?x, :type, :painter)`$\vee$ `(?x, :type, :cubist)` $\vee$ `(?x, :type, :sculptor)`

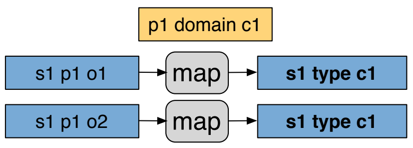
Reformulating large conjunctive queries leads to syntactically large and complex queries, for which many evaluation strategies can be devised. Such a query can be evaluated as a conjunction of unions of triple patterns, with the disadvantage of joining many intermediary results produced by each union query; or, it can be evaluated as a union of conjunctive queries, with the drawback of evaluating repeatedly those fragments which are common across the conjunctive queries. It is important to observe that such common fragments are sure to exist, since each conjunctive query features one atom from the reformulation of each initial query atoms. As a simple example, consider a conjunctive SPARQL query consisting atoms a_1, a_2 and a_3, which are reformulated respectively into the atom sets $\{a_1', a_1''\}$, $\{a_2', a_2'', a_2'''\}$, and $\{a_3', a_3''\}$. This results into $2 \times 3 \times 2 = 12$ conjunctive queries, in other words:

$$q^{ref} = \begin{array}{l} a_1', a_2', a_3' \cup a_1', a_2'', a_3' \cup a_1', a_2''', a_3' \cup \\ a_1'', a_2', a_3' \cup a_1'', a_2'', a_3' \cup a_1'', a_2''', a_3' \cup \\ a_1', a_2', a_3'' \cup a_1', a_2'', a_3'' \cup a_1', a_2''', a_3'' \cup \\ a_1'', a_2', a_3'' \cup a_1'', a_2'', a_3'' \cup a_1'', a_2''', a_3'' \end{array}$$

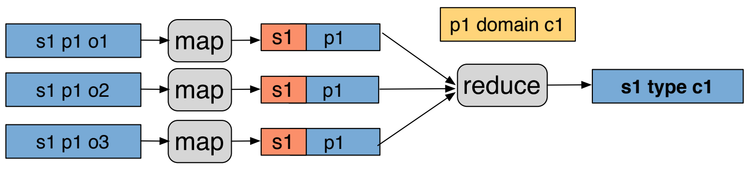
It is easy to identify pairs of conjunctive queries in the above reformulation q^{ref} sharing two atoms, e.g., a_1', a_2', a_3' and a_1'', a_2', a_3', and yet another set of union terms in q^{ref} share one atom. The repeated evaluation of such atoms is one reason why query answering through reformulation may be quite inefficient [38].

HadoopRDF [46] is the only system, among those previously mentioned, that injects some RDFS reasoning within its query processing framework; it is based on the first approach described above. More specifically, query reformulation is implicitly done during the selection of the files that must be scanned in order to start processing the query. For a given triple pattern, instead of scanning only the file that corresponds to the predicate of that pattern, the files corresponding to all the predicates occurring in the *reformulated* triple pattern are scanned and then the query is processed as described in Section 5.1. This leads to evaluating a conjunction of unions of triple patterns.

The RDFS reasoning process in [46] is based only on the RDFS *subclass hierarchy*, which means among the rules shown in Figure 1, only rules s_1 and i_2 are considered.



(a) Map-side



(b) Reduce-side

Fig. 16: Map-side and reduce-side application of rule i_3 of Table 1 when schema triples are kept in memory.

6.3 Hybrid techniques

Existing proposals from the literature combine the above reasoning approaches, that is: they precompute entailed triples for some part of the RDF data, while reformulation may still be performed at query time.

A common technique in this area is to precompute the RDFS closure of the RDF schema so that query reformulation can be made faster. This works well because the RDF schema is usually very small compared to the data, it seldom changes, and it is always used for the RDFS reasoning process. This approach is followed in [68] and [85].

Rya [68] computes the entailed triples of the RDF schema in MapReduce after loading the RDF data into the key-value store, where the RDFS closure is also stored. One MapReduce cycle is used for each level of the subclass hierarchy.

In QueryPie [85], the authors focus on the parallel evaluation of single triple pattern queries according to OWL Horst entailment rules [80], which is a superset of the RDFS entailment rules. They build and-or trees where the *or* level is used for the rules and the *and* level is used for the rules' antecedents. The root of the tree is the query triple pattern. To improve performance, entailed schema triples are precomputed so that the and-or tree can be pruned. The system is built on top of Ibis [14], a framework which facilitates the development of parallel distributed applications.

Another hybrid approach that is introduced in [48] for structured overlay networks and can be deployed in a cloud is the use of the magic sets rule rewriting algorithm [15]. The basic idea is that, given a query, rules are rewritten using information from the query so that the pre-computation of entailed triples generates only the triples required by the query. The benefit of using the new rules in the bottom-up evaluation is that it focuses only on data which is associated with the query and hence no unnecessary information is generated. Such a technique is particularly helpful in application scenarios where knowledge about the query workload is available, and therefore only the triples needed by the workload are precomputed and stored.

6.4 Summary

Table 8 summarizes the works that either are focused on or provide support for RDFS reasoning. The table spells out the reasoning method implemented in each system, the underlying framework on which reasoning takes place, the fragment of entailment rules supported and the type of queries supported for query answering (if applicable). The works [83, 84, 87], which focus on closure computation, do not consider query answering; one could deploy in conjunction with any of them, any of the query processing algorithms presented in Section 5.

7 Qualitative comparison

Concluding our previous analysis, Figure 17 shows how the systems surveyed so far can be classified according to two of the dimensions we discussed, namely data storage and query processing.

Along the data storage dimension, we identify: (i) centralized RDF stores, (ii) key-value stores and (iii) distributed file systems. With respect to query processing, we distinguish: (i) processing queries locally at one or multiple nodes (without inter-node communications), which supposes that each node has the corresponding RDF query evaluation capabilities; (ii) implementing a fully distributed evaluation engine on top of MapReduce, (iii) implementing a fully distributed evaluation engine on top of a parallel processing framework other than MapReduce, and finally (iv) based on graph partitioning and graph traversals.

Table 8: Comparison of reasoning techniques.

System	Reasoning technique	Means for reasoning	RDFS fragment	Query answering
WebPie [84]	Closure computation	MapReduce	minimal RDFS	No
Weaver et al. [87]	Closure computation	MPI	full RDFS	No
WebPie [83]	Closure computation	MapReduce	OWL Horst	No
HadoopRDF [46]	Query reformulation	Pellet reasoner to find input files	RDFS subclass only	Conjunctive
Rya [68]	Hybrid	MapReduce	RDFS subclass only	Conjunctive
QueryPie [85]	Hybrid	Ibis framework	OWL Horst	Single triple patterns

The most popular options taken rely on DFS and MapReduce; combining key-value stores with local query processing is also quite popular. This is due to the facility with which these architectures allow to store and process data in large-scale distributed platforms, without the application programmers having to handle issues such as fault-tolerance and scalability. However, there are many combinations of choices (of storage and processing) which could still be investigated.

Each option has its trade-offs. For instance, key-value stores offer a fine-granularity indexing mechanism allowing very fast triple pattern matching, however, they do not rival the parallelism offered by MapReduce for processing efficiently queries, and thus, most systems perform joins locally at a single site. Although this approach may be efficient for very selective queries with few intermediate results, it is not scalable for analytical-style queries which need to access big portions of RDF data. For the latter, MapReduce is more appropriate, especially if a fully parallel query plan is employed. Approaches based on centralized RDF stores are well-suited for star-join queries, since triples sharing the same subject are typically grouped on the same site. Thus, the centralized RDF engine available on that site can be leveraged to process efficiently the query for the respective data subset; overall, such queries are efficiently evaluated by the set of single-site engines working in parallel. In contrast, path queries which need to traverse the subsets of the RDF graph stored at distinct sites involve more communications between machines and thus their evaluation is less efficient. Finally, it may be worth noting that a parallel processor built out of a set of single-site ones leaves open issues such as fault tolerance and load balancing, issues which are implicitly handled by frameworks such as MapReduce.

Since each option of storage and processing has their pros and cons, some works follow a hybrid solution fitting more than one cell in our table. For instance, H_2RDF [66] and Huang et al. [44] adopt both a local and a MapReduce-based query evaluation, while AMADA takes advantage of all three storage facilities. Considering the variety of requirements (point queries versus large analytical ones, star versus chain queries, updates etc.), a combination of techniques, perhaps with some adaptive techniques taking into account the characteristics of a particular RDF data set and workload, is likely to lead to the best performance overall.

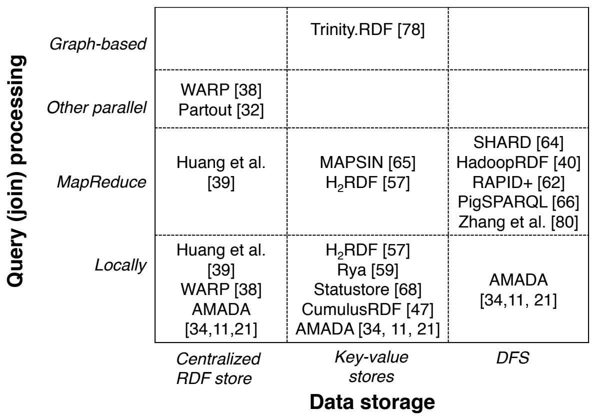


Fig. 17: System classification in the two dimensions.

Table 9 summarizes the technical characteristics of each system, such as: the underlying framework required to run, the fragment of SPARQL it supports, whether reasoning is supported, and the link to the source code, whenever this is available.

8 Conclusions and open future directions

RDF has been successfully used to encode semi-structured data in a variety of application contexts [82]; in particular, an interesting class of RDF applications comes from the area of Linked Data, where data sets independently produced can be interconnected and interpreted together based on their usage of common resource identifiers (or URIs). Further, data sets made for sharing are quite often endowed with explicit schema descriptions, allowing a third party to interpret and best exploit the data. RDF natively allows simple yet expressive and flexible schema descriptions by means of RDF Schema (RDFS) statements, making it an ideal candidate for data interoperability.

Efficiently processing large volumes of RDF data defeats the possibilities of a single centralized system. In this context, recent research has sought to take advantage of the large-scale parallelization possibilities provided by the cloud, while also enjoying features such as automatic scale-up and scale-down of resource allocation as well as some level of resilience to system failures.

Table 9: Technical aspects of the systems.

System	Underlying framework	SPARQL fragment	Reasoning	Source code
SHARD [72]	Hadoop	BGP	-	`http://sourceforge.net/projects/shard-3store/`
HadoopRDF [46, 45]	Hadoop	BGP	✓	`https://code.google.com/p/hadooprdf/`
PigSPARQL [75]	Hadoop/Pig	Full SPARQL 1.0	-	-
RAPID+ [70, 51, 52]	Hadoop	BGP	-	-
Zhang et al. [93]	Hadoop	BGP	-	-
MAPSIN [76]	Hadoop/HBase	BGP	-	-
H_2RDF [66]	Hadoop/HBase	BGP	-	`http://code.google.com/p/h2rdf/`
EAGRE [92]	Hadoop	BGP+range filters	-	-
Rya [68]	Accumulo	BGP+time ranges	✓	-
Stratustore [78]	SimpleDB	BGP	-	`http://code.google.com/p/stratustore/`
CumulusRDF [55]	Cassandra	Single triple pattern	-	-
AMADA [21, 11, 22]	AWS	BGP	-	`http://cloak.saclay.inria.fr/research/amada/`
Huang et al. [44]	Hadoop/RDF-3X	BGP	-	-
WARP [43]	RDF-3X	BGP	-	-
Partout [37]	RDF-3X	BGP	-	-
Wu et al. [90]	HadoopDB	BGP	-	-
Trinity.RDF [91]	Trinity	BGP	-	-

In this survey, we have focused on presenting the state-of-the-art in the area of RDF data management in distributed or parallel settings; some of the systems we survey have been built specifically on the primitives of a commercial cloud provider while others can be seen as distributed data management systems that can be deployed in a cloud (but also in a large-scale cluster architecture). We classified the systems according to the way in which they implement three fundamental functionalities: *data storage*, *query processing*, and *reasoning*, detailed the existing solutions adopted to implement each of these, and classified the existing systems in the dimension space thus obtained. We observed a great density of systems using MapReduce and DFS, as well as NoSQL systems with local clients for performing more complex tasks, presumably because such underlying infrastructures are very easy to use. We expect more mature systems to be developed in the near future which may combine these two building blocks or are based on their own infrastructure.

We currently find numerous open problems as the research area of parallel RDF data management is only a few years mature. Firstly, an important issue that needs to be investigated concerns optimization techniques for statistics gathering, query decomposition, and join ordering. Techniques from the existing literature on distributed data management [65] can and should be adopted, to improve processing performance and overall resource management.

RDFS reasoning is an essential functionality of the RDF data model and it needs to be taken into account for RDF stores to provide correct and complete query answers. While some works investigated the parallelization of the RDFS closure computation, the area of query reformulation is so far unexplored in a parallel environment for conjunctive RDF queries. Query reformulation can benefit from techniques for multi-query optimization like the ones proposed in [86, 34] and scan sharing [52], but adapting such techniques for RDF data is not straightforward.

In addition, current works focus only on the conjunctive fragment of SPARQL. Although this is the first step towards RDF query processing, SPARQL allows for much more expressive queries, e.g., queries including optional clauses, aggregations and property paths[3]. New frameworks for RDF style analytics have appeared [26], which may naturally be adapted to a large-scale (cloud) context. Evaluating such queries in a parallel environment is still an open issue.

As RDF data volumes increase, future systems should be able to use only parts of RDF data to answer user queries. This can be achieved by building indexes specific to RDF for routing queries to specific data sets [11], or reducing I/O [92], or by maintaining views on query results to be used in subsequent requests. We believe these topics will attract significant interest in the near future.

References

1. D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.*, 18(2):385–406, 2009.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Addison-Wesley, 1995.
3. S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE*, pages 606–615, 2008.
4. A. Abouzeid, Bajda-Pawlikowski, D. K., Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB*, 2009.

[3] `http://www.w3.org/TR/sparql11-property-paths/`

5. F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Trans. Knowl. Data Eng.*, 23(9), 2011.
6. Foto N. Afrati and Jeffrey D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT*, 2010.
7. Apache Accumulo. `http://accumulo.apache.org/`, 2012.
8. Apache Cassandra. `http://cassandra.apache.org/`, 2012.
9. Apache Hadoop. `http://hadoop.apache.org/`, 2012.
10. Apache HBase. `http://hbase.apache.org/`, 2012.
11. A. Aranda-Andújar, F. Bugiotti, J. Camacho-Rodríguez, D. Colazzo, F. Goasdoué, Z. Kaoudi, and I. Manolescu. Amada: Web Data Repositories in the Amazon Cloud (demo). In *CIKM*, 2012.
12. M. Arias, J.D. Fernández, M.A. Martínez-Prieto, and P. de la Fuente. An Empirical Study of Real-World SPARQL Queries. In *USEWOD*, 2011.
13. Amazon Web Services. `http://aws.amazon.com/`, 2012.
14. H. E. Bal, J. Maassen, R. V. van Nieuwpoort, N. Drost, R. Kemp, N. Palmer, G. Wrzesinska, T. Kielmann, F. Seinstra, and C. Jacobs. Real-World Distributed Computing with Ibis. *IEEE Computer*, 43(8):54–62, 2010.
15. F. Bancilhon, D. Maier, Y. Sagiv, and J. D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS*, 1986.
16. T. Berners-Lee. Linked data - design issues. `http://www.w3.org/DesignIssues/LinkedData.html`, 2006.
17. S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *SIGMOD*, 2010.
18. M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *SIGMOD*, pages 121–132, 2013.
19. D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, W3C Recommendation, 2004.
20. J. Broekstra and A. Kampman. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC*, 2002.
21. F. Bugiotti, J. Camacho-Rodríguez, F. Goasdoué, Z. Kaoudi, I. Manolescu, and S. Zampetakis. SPARQL Query Processing in the Cloud. In A. Harth, K. Hose, and R. Schenkel, editors, *Linked Data Management*. Chapman and Hall/CRC, 2014.
22. F. Bugiotti, F. Goasdoué, Z. Kaoudi, and I. Manolescu. RDF Data Management in the Amazon Cloud. In *DanaC Workshop (in conjunction with EDBT)*, 2012.
23. R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, May 2011.
24. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI*, 2006.
25. E. Inseok Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB*, 2005.
26. D. Colazzo, F. Goasdoué, I. Manolescu, and A. Roatiş. RDF Analytics: Lenses over Semantic Graphs. In *WWW*, 2014.
27. T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. Mapreduce online. In *NSDI*, 2010.
28. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
29. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.
30. J. Dittrich, J.-A. Quiane-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). In *PVLDB*, pages 518–529, 2010.
31. J. Dittrich, J.-A. Quiane-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only aggressive elephants are fast elephants. In *PVLDB*, pages 1591–1602, 2012.
32. C. Doulkeridis and K. Norvag. A survey of large-scale analytical query processing in MapReduce. *The VLDB Journal*, 2013.
33. DynamoDB. `http://aws.amazon.com/dynamodb/`.
34. I. Elghandour and A. Aboulnaga. ReStore: Reusing Results of MapReduce Jobs. *PVLDB*, 5(6):586–597, 2012.
35. O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. *Networked Knowledge - Networked Media*, pages 7–24, 2009.
36. I. Filali, F. Bongiovanni, F. Huet, and F. Baude. A Survey of Structured P2P Systems for RDF Data Storage and Retrieval. *T. Large-Scale Data- and Knowledge-Centered Systems*, 3:20–55, 2011.
37. L. Galarraga, K. Hose, and R. Schenkel. Partout: A Distributed Engine for Efficient RDF Processing. Technical Report: CoRR abs/1212.5636, 2012.
38. F. Goasdoué, I. Manolescu, and A. Roatiş. Efficient query answering against dynamic RDF databases. In *EDBT*, 2013.
39. W3C OWL Working Group. OWL 2 Web Ontology Language. W3C Recommendation, December 2012. `http://www.w3.org/TR/rdf-mt/`.

40. S. Harris, N. Lamb, and N. Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. In *SSWS Workshop*, 2009.
41. S. Harris and A. Seaborne. SPARQL 1.1 Query Language. W3C Recommendation, `http://www.w3.org/TR/sparql11-overview/`, 2013.
42. P. Hayes. RDF Semantics. W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-mt/`.
43. K. Hose and R. Schenkel. WARP: Workload-Aware Replication and Partitioning for RDF. In *DESWEB Workshop (in conjunction with ICDE)*, 2013.
44. J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.
45. M. Husain, L. Khan, M. Kantarcioglu, and B. M. Thuraisingham. Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools. In *IEEE CLOUD*, pages 1–10, 2010.
46. M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. on Knowl. and Data Eng.*, 2011.
47. J.K.Lawder and P.J.H.King. Using Space-filling Curves for Multi-Dimensional Indexing. In *British National Conference on Databases: Advances in Databases*, 2000.
48. Z. Kaoudi and M. Koubarakis. Distributed RDFS Reasoning over Structured Overlay Networks. *Journal on Data Semantics*, 2013.
49. Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou, and A. Papadakis-Pesaresi. Atlas: Storing, updating and querying RDF(S) data on top of DHTs. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4), 2010.
50. Z. Kaoudi, K. Kyzirakos, and M. Koubarakis. SPARQL Query Optimization on Top of DHTs. In *ISWC*, 2010.
51. H. Kim, P. Ravindra, and K. Anyanwu. From SPARQL to MapReduce: The Journey Using a Nested Triple-Group Algebra (demo). *PVLDB*, 4(12):1426–1429, 2011.
52. H. Kim, P. Ravindra, and K. Anyanwu. Scan-Sharing for Optimizing RDF Graph Pattern Matching on MapReduce. In *IEEE Conference on Cloud Computing*, pages 139–146, 2012.
53. A. Kiryakov, B. Bishoa, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. The Features of BigOWLIM that Enabled the BBC's World Cup Website. In *Workshop on Semantic Data Management*, 2010.
54. G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 2004.
55. G. Ladwig and A. Harth. CumulusRDF: Linked Data Management on Nested Key-Value Stores. In *SSWS*, 2011.
56. State of the LOD cloud. Available from: `http://www4.wiwiss.fu-berlin.de/lodcloud/state/`, 2011.
57. F. Manola and E. Miller. RDF Primer. W3C Recommendation, February 2004.
58. METIS. http://glaros.dtc.umn.edu/gkhome/views/metis.
59. S. Muñoz, J. Pérez, and C. Gutierrez. Simple and Efficient Minimal RDFS. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):220–234, September 2009.
60. T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDBJ*, 19(1), 2010.
61. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.
62. K. Ono and G. M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *VLDB*, pages 314–325, 1990.
63. Marin Dimitrov (Ontotext). Semantic technologies from big data. `http://www.slideshare.net/marin_dimitrov/semantic-technologies-for-big-data`, 2012.
64. A. Owens, A. Seaborne, N. Gibbins, and M. Schraefel. Clustered TDB: A Clustered Triple Store for Jena. Technical Report, 2008.
65. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer, 2011.
66. N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H_2RDF: adaptive query processing on RDF data in the cloud (demo). In *WWW*, 2012.
67. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transaction Database Systems*, 34:16:1–16:45, September 2009.
68. R. Punnoose, A. Crainiceanu, and D. Rapp. Rya: A Scalable RDF Triple Store for the Clouds. In *Workshop on Cloud Intelligence (in conjunction with VLDB)*, 2012.
69. G. Raschia, M. Theobald, and I. Manolescu. Proceedings of the first International Workshop On Open Data (WOD), 2012.
70. P. Ravindra, H. Kim, and K. Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *ESWC*, pages 46–61, 2011.
71. K. Rohloff and R. E. Schantz. High-Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: the SHARD Triple-Store. In *Programming Support Innovations for Emerging Distributed Applications*, 2010.

72. K. Rohloff and R. E. Schantz. Clause-Iteration with MapReduce to Scalably Query Datagraphs in the SHARD Graph-Store. In *Workshop on Data-intensive Distributed Computing*, 2011.
73. S. Sakr, A. Liu, and A. G. Fayoumi. The Family of Mapreduce and Large-scale Data Processing Systems. *ACM Comput. Surv.*, 46(1):11:1–11:44, 2013.
74. M. Saleem, M. R. Kamdar, A. Iqbal, S. Sampath, H. F. Deus, and A. Ngonga. Fostering Serendipity through Big Linked Data. In *Semantic Web Challenge at ISWC*, 2013.
75. A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *SWIM*, 2011.
76. A. Schätzle, M.n Przyjaciel-Zablocki, C. Dorner, T. Hornung, and G. Lausen. Cascading Map-Side Joins over HBase for Scalable Join Processing. In *SSWS+HPCSW*, 2012.
77. B. Shao, H. Wang, and Y. Li. The Trinity Graph Engine. Technical report, `http://research.microsoft.com/pubs/161291/trinity.pdf`, 2012.
78. R. Stein and V. Zacharias. RDF On Cloud Number Nine. In *Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic*, May 2010.
79. The Cancer Genome Atlas project. `http://cancergenome.nih.gov/`.
80. H. J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Web Semantics*, 3(2-3):79–115, 2005.
81. Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In *ISWC*, 2005.
82. S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, 2007.
83. J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. E. Bal. OWL Reasoning with WebPIE: Calculating the Closure of 100 Billion Triples. In *ESWC*, pages 213–227, 2010.
84. J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable Distributed Reasoning using MapReduce. In *ISWC*, 2009.
85. J. Urbani, F. van Harmelen, S. Schlobach, and H. Bal. QueryPIE: Backward Reasoning for OWL Horst over Very Large Knowledge Bases. In *ISWC*, 2011.
86. G. Wang and C. Chan. Multi-Query Optimization in MapReduce Framework. *PVLDB*, 7(3):145–156, 2013.
87. J. Weaver and J. A. Hendler. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In *ISWC*, 2009.
88. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.
89. K. Wilkinson, C. Sayers, H. A. Kuno, and D. Raynolds. Efficient RDF Storage and Retrieval in Jena2. In *SWDB (in conjunction with VLDB)*, 2003.
90. B. Wu, H. Jin, and P. Yuan. Scalable SAPRQL Querying Processing on Large RDF Data in Cloud Computing Environment. In *ICPCA/SWS*, pages 631–646, 2012.
91. K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A Distributed Graph Engine for Web Scale RDF Data. In *PVLDB*, 2013.
92. X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: Towards Scalable I/O Efficient SPARQL Query Evaluation on the Cloud. In *ICDE*, 2013.
93. X. Zhang, L. Chen, and M. Wang. Towards Efficient Join Processing over Large RDF Graph Using MapReduce. In *SSDBM*, pages 250–259, 2012.