© 2014 by the authors; licensee RonPub, Lübeck, Germany. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/3.0/).

Open Access

Open Journal of Semantic Web (OJSW)
Volume 1, Issue 1, 2014

http://www.ronpub.com/ojsw
ISSN 2199-336X

MapReduce-based Solutions for Scalable SPARQL Querying

José M. Giménez-García [A], Javier D. Fernández [A], Miguel A. Martínez-Prieto [B]

[A] DataWeb Research, Department of Computer Science, E.T.S.I.I., University of Valladolid.
Campus Miguel Delibes, 47011, Valladolid, Spain.
josemiguel.gimenez@alumnos.uva.es, jfergar@infor.uva.es

[B] DataWeb Research, Department of Computer Science, E.U.I., University of Valladolid.
Campus María Zambrano, 40005, Segovia, Spain.
migumar2@infor.uva.es

Abstract

The use of RDF to expose semantic data on the Web has seen a dramatic increase over the last few years. Nowadays, RDF datasets are so big and interconnected that, in fact, classical mono-node solutions present significant scalability problems when trying to manage big semantic data. MapReduce, a standard framework for distributed processing of great quantities of data, is earning a place among the distributed solutions facing RDF scalability issues. In this article, we survey the most important works addressing RDF management and querying through diverse MapReduce approaches, with a focus on their main strategies, optimizations and results.

Type of Paper and Keywords

Regular research paper: *MapReduce, RDF Graphs, SPARQL*

1 Introduction

Massive publication efforts have flooded the Web with huge amounts of semantic data represented in RDF [37]. The *Resource Description Framework* (RDF) provides a graph-based model to structure and link data along the so-called Semantic Web. An RDF dataset draws a directed labeled graph of knowledge in which entities and values are linked via labeled edges with meaning.

In less than a decade, RDF has become a popular format to expose and interlink data from almost every field of knowledge, from bioinformatics and geography to social networks. There are two main reasons for this success. First, its semantic model is extremely simple: entities (also called resources) are described in the form of triples `(subject, predicate, object)`. Then, RDF itself is schema-relaxed and its vocabulary can evolve as needed. Furthermore, very active Open Data initiatives, such as the Linked Open Data[1] (LOD) community, are promoting its use to publish structured data on the Web and to connect it to other data sources [5].

In turn, SPARQL [45] is the W3C recommendation to search and extract information from RDF graphs. SPARQL is essentially a declarative language based on graph-pattern matching with SQL-like syntax. Graph patterns are built on top of *Triple Patterns* (triples allowing variable components) which are grouped within *Basic Graph Patterns*, leading to query subgraphs in which variables must be bounded. Graph patterns commonly *join* triple patterns, but other constructions are also possible (see Section 2.2). Thus, query resolution performance mainly depends on two factors: (i) *retrieving RDF triples* (for triple pattern resolution) de-

[1] `http://linkeddata.org/`

pends on how triples are organized, stored, and indexed. Then, (ii) *join* performance is determined by optimization strategies [51], but also by the join resolution algorithms. Both factors are typically addressed within RDF storage systems, referred to as *RDF stores*.

RDF stores [10] are built on top of relational systems or are carefully designed from scratch to fit particular RDF peculiarities. In any case, current RDF stores typically lack scalability when large volumes of RDF data must be managed [15]. This fact hinders the construction of scalable applications in the increasingly large LOD cloud. As evidence of this RDF growth, LOD-Stats[2], a project constantly monitoring the LOD cloud, reported that the number of triples grew to more than 62 billion triples in November 2013. A more detailed analysis shows that many single datasets comprise more than a hundred million triples, and the storage of such datasets in a mono-node machine, which must provide efficient access, becomes a formidable challenge. Unfortunately, while significant progress has been made on RDF storage on a mono-node configuration, such as highly-compressed indexes [38], distributed solutions are still at initial stages and continue to be an open challenge [25].

In this scenario, MapReduce [11] arises as a candidate infrastructure for dealing with the efficient processing of big semantic datasets. In short, MapReduce is a framework for the distributed processing of large quantities of data. Initially developed by Google [11], it is widely used in environments of Big Data processing [60]. Data from Yahoo! [53], Facebook [56] or Twitter [34] are managed using MapReduce. Obviously, it has also been tested for RDF management at large scale. In this paper, we survey the state of the art concerning MapReduce-based solutions providing an efficient resolution of SPARQL on a large scale . Our main goal is to gain insights about how current MapReduce-based solutions store RDF and perform SPARQL resolution – their main decisions and strategies –, and to analyze their opportunities in the near future, in which large-scale processing will be even more demanding.

The rest of the paper is organized as follows. Section 2 provides basic notions about RDF and SPARQL, whereas Section 3 describes MapReduce foundations to provide basic knowledge about its strengths, but also its weaknesses. In Section 4, we depict the main challenges which arise when MapReduce is used to meet the needs of RDF storage and SPARQL resolution. Then, in Section 5, we introduce a simple classification in which the most-prominent techniques are organized according to the complexity that their nodes assume within MapReduce clusters. This organization is used for explaining and analyzing these techniques in Sections 6 and 7, and their experimental results are studied in Section 8. These results discover the current limitations, and help us to reach conclusions about the interesting research opportunities arising in SPARQL resolution on a large scale under MapReduce-guided scenarios. These opportunities are finally discussed in Section 9.

[2] `http://stats.lod2.eu/`

2 RDF & SPARQL

The Semantic Web emerges over a technology stack comprising a great variety of standards, formats, and languages. Nevertheless, this paper focuses on how data is effectively stored and represented for query resolution, hence our interest is exclusively related to the aforementioned RDF and SPARQL.

2.1 RDF

RDF is the data model used for semantic data organization. It is a directed labeled graph in which resources are described through properties and the values for these properties. Triples (also called statements) are the atoms of RDF: `(subject, predicate, object)`. The subject identifies the resource being described, the predicate is the property applied to it, and the object is the concrete value for this property. Figure 1 shows an RDF excerpt with five RDF triples (left) and its graph-based representation (right). As can be seen, part of the success of RDF is due to this direct labeled graph conception and its expressive power: an RDF dataset models a network of statements through natural relationships between data, by means of labeled edges. The labeled graph structure underlying the RDF model allows new semantics to be easily added on demand. In other words, graph flexibility allows semi-structured information (entities having different levels of detail) to be handled.

As shown in the previous example, the RDF data model considers that subjects (the resources being described) and predicates are identified by URIs [4], whereas objects (the values for the properties) can be either other resources or constant values (called *literals*). There exists a special kind of node called *blank node* which identifies unnamed resources, usually serving as parent nodes of a group of triples (containers, collections, etc.). Thus, the RDF data model is typically formalized as follows [22]. Assume infinite, mutually disjoint sets U (*RDF URI references*), B (*Blank nodes*), and L (*RDF literals*).

Definition 1 (RDF triple) *A tuple $(s,p,o) \in (U \cup B) \times U \times (U \cup B \cup L)$ is called an RDF triple, in which "s" is the subject, "p" the predicate and "o" the object.*

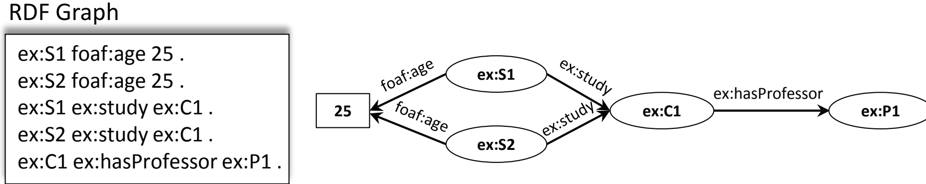

Figure 1: An RDF graph example.

SPARQL Query

```
SELECT ?age ?courses
WHERE {
  ex:S1 foaf:age ?age .
  ex:S1 ex:study ?courses .
}
```

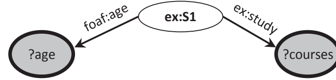

Figure 2: A first SPARQL query.

Definition 2 (RDF graph) *An RDF graph G is a set of RDF triples. As stated, (s,p,o) can be represented as a direct edge-labeled graph $s \xrightarrow{p} o$.*

2.2 SPARQL

SPARQL is the W3C standard to query the RDF data model. It follows the same principles (interoperability, extensibility, decentralization, etc.) and a similar graph notion. Intuitively, a SPARQL graph pattern comprises named terms but also variables for unknown terms. The pattern returns solutions when it matches an RDF subgraph after variable substitution. This required substitution of RDF terms for the variables is then the solution for the query.

An SPARQL query example, with the appropriate syntax, is presented on the left side of the Figure 2, whereas the right side represents the intuitive query graph pattern. The WHERE clause serializes the graph pattern to be matched against the RDF graph, whereas the SELECT clause lists those variables delivered as results. In the current example, when the aforementioned query is matched to the RDF graph in Figure 1, the result is a simple *mapping*: `?age="25"` and `?courses= ex:C1`.

The smaller components of a graph pattern are *Triple Patterns* (hereafter *TPs*), *i.e.*, triples in which each of the subject, predicate and object may be a variable (this is formalized in Definition 3). The previous example is a *Basic Graph Pattern* (hereafter *BGP*) composed of two joined triple patterns. In general terms, BGPs are sets of triple patterns in which all of them must be matched (this is formalized in Definition 4). They can be seen as inner-joins in SQL.

Formally speaking, in SPARQL, URIs are extended to IRIs [12], hence we introduce a set I of RDF IRI references, and a novel set, V of variables, disjoint from I. Thus, following Pérez et al. [44], an **RDF triple** is now a statement $(s,p,o) \in (I \cup B) \times I \times (I \cup B \cup L)$, and we can formally define TPs and BGPs, as follows:

Definition 3 (SPARQL triple pattern) *A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a triple pattern.*

Note that blank nodes act as non-distinguished variables in graph patterns [45].

Definition 4 (SPARQL Basic Graph pattern (BGP)) *A SPARQL Basic Graph Pattern (BGP) is defined as a set of triple patterns. SPARQL FILTERs can restrict a BGP. If B_1 is a BGP and R is a SPARQL* built-in *condition, then $(B_1$ `FILTER` $R)$ is also a BGP.*

Several constructors can be applied over BGPs, such as the `UNION` of two groups (similar to SQL), `OPTIONAL` graph patterns (similar to the relational left outer join [44]) or `FILTER` conditions to provide restrictions on solutions (such as *regex* functions). Pérez et al. [44] complete this formalization with more semantics (mappings, evaluation, etc.) and a deep study on complexity query evaluation. Anglés and Gutiérrez [2] reveal that the SPARQL algebra has the same expressive power as Relational Algebra, although their conversion is not trivial [9].

3 MAPREDUCE

MapReduce [11] is a framework and programming model for the distributed processing of large volumes of data. In practice, it can be considered as the *de facto* standard for Big Data processing. Its main aim is to provide parallel processing over a large number of nodes. MapReduce takes autonomous charge of the parallelization task, data distribution and load balancing between every node, as well as fault tolerance on the cluster [60]. This enables a wide range of parallel applications in which developer responsibilities focus, exclusively, on designing the specific problem solution.

MapReduce is not schema-dependent, so it can process unstructured and semi-structured data at the price of parsing every input item [35]. MapReduce relies on two main operations which process data items in the form of *key-value* pairs: `Map` and `Reduce`. `Map` tasks read input pairs, and generate lists of new pairs comprising

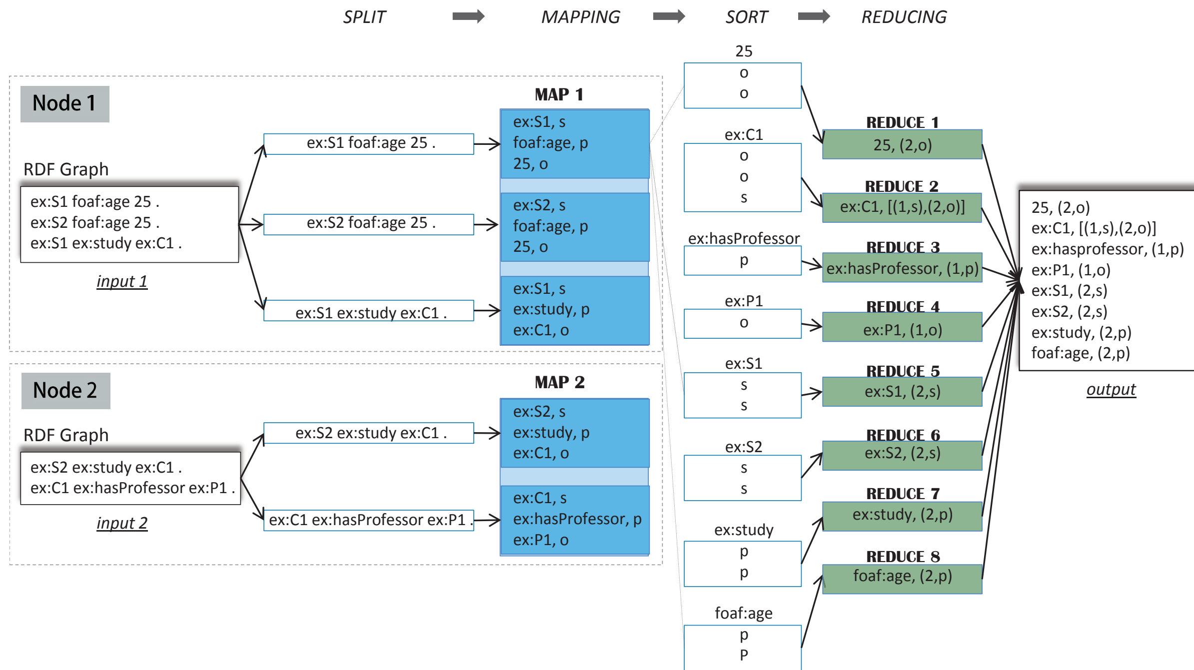

Figure 3: The overall MapReduce count resources process.

all different values gathered for a given key. Note that the domain of input keys and values is different from the domain of output keys and values (*i.e.*, input pairs correspond to raw data read, and output pairs correspond to processed data). These lists are sent to the `Reduce` tasks by an intermediate step. A `Reduce` task reads its inputs, groups them, and obtains final values. In this way, its input and output keys and values belong to the same domain. A generic example of input and output types is illustrated in Listing 1 [11]. It is worth emphasizing that `Map` and `Reduce` tasks perform exhaustive I/O disk operations. Both `Map` and `Reduce` read inputs from disk and their results are written again on disk. This fact turns I/O disk operations into the main bottleneck of MapReduce processes [35], and must be taken into consideration when designing an optimized MapReduce process [28].

Listing 1: `Map` and `Reduce` operations diagram

$$\begin{array}{ll} \text{map:} & (k1, v1) \rightarrow list(k2, v2) \\ \text{reduce:} & (k2, list(v2)) \rightarrow list(v2) \end{array}$$

A MapReduce cluster has a Master/Slave architecture. A single *Master* initializes the process, schedules tasks and keeps bookkeeping information. All other nodes are *Workers*, which run `Map` and `Reduce` tasks [11]. Typically, a single machine processes several tasks [60]. Data are stored in a distributed manner among nodes, divided into file chunks of predefined size (typically 64 or 128 MB). MapReduce takes advantage of *data locality*, so each `Map` task reads data from the same machine it is running on, thus avoiding unnecessary bandwidth usage [11]. When `Map` tasks end, their output is divided into as many chunks as keys, and each one is sent to its corresponding `Reduce` node. Final results of `Reduce` tasks are distributed again among the cluster machines, as they are commonly used in a subsequent MapReduce job [60].

Example. Let us suppose that we design a MapReduce-based solution to process the simple RDF excerpt illustrated in Figure 1. Our objective is to count the number of occurrences of different resources within the RDF dataset, and group them by attending to their role in the graph (subject, predicate, or object). The input data are assigned to the `Map` tasks, attempting to provide them with the chunk residing in the same machine. Each `Map` task reads its assigned triples, one by one, and outputs three pairs for each triple, stating the resource and the role it plays: `(<resource_subject>,s)`, `(<resource_predicate>,p)`, and `(<resource_object>,o)`.

The `Map` output is grouped by resource and sent to different `Reduce` tasks, one per different resource. Each `Reduce` increments the number of occurrences by role and writes the results on disk. This process is illustrated

in Figure 3. As can be seen, MapReduce is deployed using two nodes, hence the original RDF dataset is divided into two chunks. Node 1 processes the first three triples, and Node 2 works with the remaining two. Each `Map` task outputs its corresponding pairs, which are then sorted before the `Reduce` tasks, in which the final statistics are obtained. □

3.1 Hadoop & Related Technologies

Apache Hadoop[3] provides the most relevant implementation of the MapReduce framework, published under the Apache License 2.0[4]. It is designed to work in heterogeneous clusters of commodity hardware with Unix/Linux operating systems. To the best of our knowledge, Hadoop is the implementation used in each academic work proposing MapReduce for SPARQL resolution.

Hadoop implements *HDFS* (Hadoop Distributed File System) [6], a distributed file system based on the Google File System model [18]. It is designed to provide scalability, fault tolerance and high aggregate performance. An important feature of HDFS is data replication: each file is divided into blocks and replicated among nodes (the replication factor in HDFS is three by default). This replication also favors the aforementioned *data locality*, *i.e.* `Map` tasks are performed in the same node where their input data are stored.

Replicated data files in HDFS are schema-free and index free, so each content must be parsed explicitly. In practice, this fact overloads MapReduce performance, hence different alternatives can be considered to minimize this impact. One of the most relevant alternatives is *HBase*[5]: this is a NoSQL column-oriented database which takes advantage of the distributed HDFS features and provides Google BigTable [8] capabilities on top of HDFS and Hadoop. HBase indexes data by row key, column key, and a timestamp value, and maps them into an associated byte array which allows random data access to MapReduce applications. HBase also allows secondary indexes and filters that reduce data transferred over the network, improving the overall MapReduce performance.

4 Challenges for MapReduce-based SPARQL resolution

As explained in Section 2.2, SPARQL resolution usually takes the form of pattern matching between the RDF graph and the BGP [25]. In distributed environments, each node is responsible for matching the corresponding query against the subgraph that it stores. If queries are simple TPs, each node obtains its partial result set which is then merged with the result sets retrieved from the other nodes. However, general SPARQL queries (in the form of conjunctive TPs) imply cross joins between different nodes. This is a traditional complex operation in distributed databases, where node coordination is mandatory and data exchanging between them overloads the overall query performance. MapReduce-based approaches follow the aforementioned `Map` and `Reduce` job to face this problem in the SPARQL ecosystem.

In a MapReduce cluster, **the full RDF dataset is distributed between the nodes**. When a new query is received, each node runs a `Map` task on its corresponding data and **obtains all the triples satisfying the query locally**. These temporary results are sorted by the corresponding join variable and sent to `Reduce` tasks, which finally **perform the join**. Note that these tasks have no communication with each other, and read different result chunks.

A naive approach, with no optimizations, would follow the next general steps:

1. In the `Map` stage, triples are read from their storage and transformed into key-value pairs in which keys correspond to the join variable bindings. Note that the process is performed for the two TPs in the join. These results are written to disk.

2. Those retrieved triples are read in the form of key-value pairs and sorted by key before `Reduce`.

3. The `Reduce` stage reads the pairs and keeps those matching the join condition. Obviously, two pairs join when they share the same key. These triples are written in the final output.

It is worth noting that some joins can depend on other unresolved joins, *i.e.* when a TP is used in two joins over different variables [26], and triples can be scattered in an unknown way over the data excerpts. Thus, it is possible that more than one job is needed to obtain the final result. In such a scenario, these jobs perform sequentially, and each one takes the previous job output as input [28]. In this case, a first step is mandatory to select which variables will be resolved in each job [26, 39].

A natural straightforward optimization is to implement **multi-way join resolution** [24] in MapReduce, *i.e.*, the possibility of performing a join on more than two TPs sharing the same variable. In other words, MapReduce allows multi-way joins to be resolved in a single job [1], in contrast to classical solutions which make as many two-way joins as needed [39]. This can be useful to resolve the so-called *Star Queries* (several TPs

[3] `http://hadoop.apache.org`
[4] `http://www.apache.org/licenses/LICENSE-2.0`
[5] `https://hbase.apache.org`

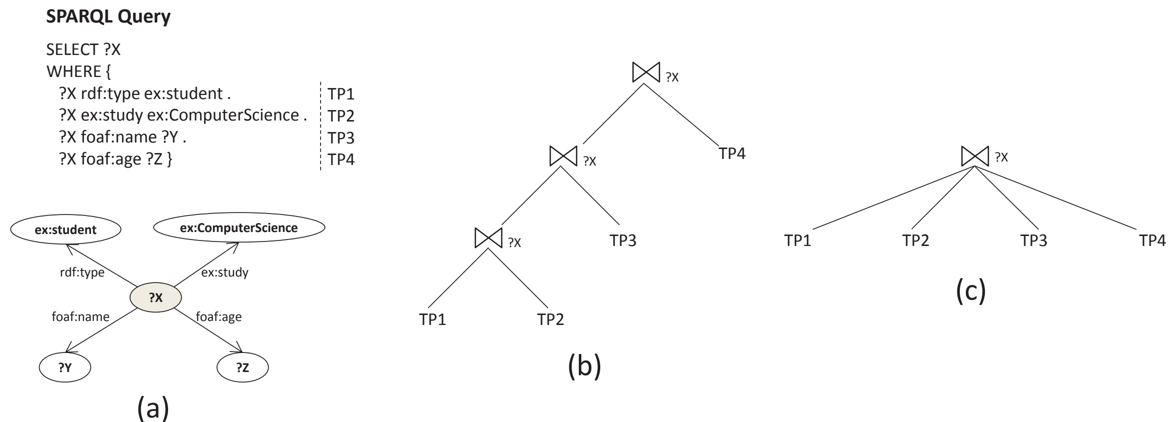

Figure 4: Potential execution plans in SPARQL star-shaped queries. (a) Original query, (b) Plan constructed by traditional two-ways joins, (c) Optimized plan with multi-way joins.

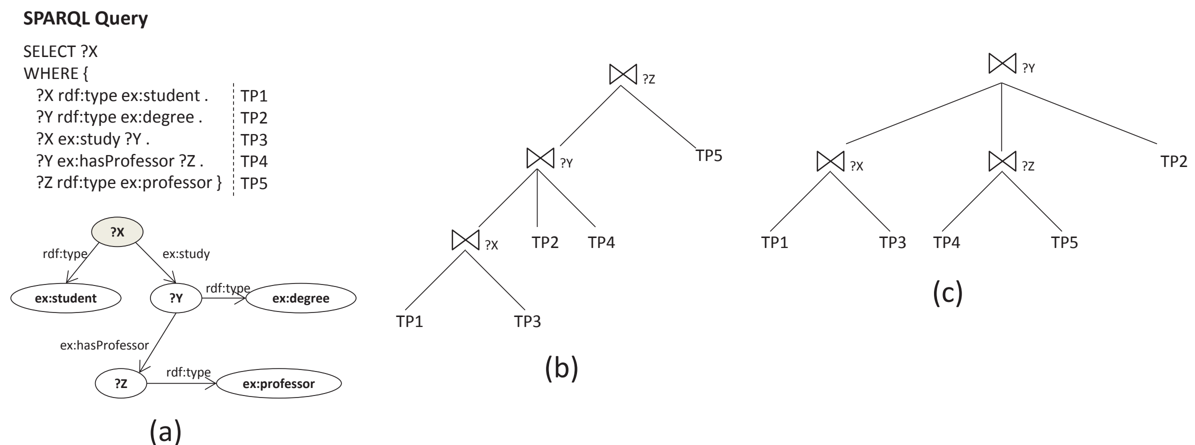

Figure 5: Potential execution plans in SPARQL path queries. (a) Original query, (b) Plan constructed with multi-way joins, (c) Optimized plan with multi-way joins.

sharing a join variable), such as the query illustrated in Figure 4(a).

The classical resolution plan for this query, shown in Figure 4(b), would make a first join between `TP1` and `TP2`, a second join between the previous result set and `TP3`, and finally a join between the output of the previous one and `TP4`. Otherwise, this could also be solved by means of two independent joins, for instance `TP1`⋈`TP2` and `TP3`⋈`TP4`, and a final join of both results. In contrast, MapReduce can make all these joins in just one step [39], because all TPs share the same variable. This is illustrated in Figure 4(c).

In general, the exact number of required MapReduce jobs depends on the order in which join variables are chosen. In addition, each job supposes a serious overhead, which can be almost equivalent to reading a billion triples [27]; partly because of job initialization [43], partly because of data shuffling between `map` and `reduce` tasks [49, 19]. For this reason, **efficient variables selection** is crucial [28, 26]. It can be seen in the *Path Query* (a query in which TPs form a chained path) described in Figure 5(a). This query can be resolved with the resolution plan illustrated in Figure 5(b). A first job can produce the join between `TP1` and `TP3`, over `?X`. Then, a multi-way join between the result of this first job, `TP2`, and `TP4`, over `?Y`, can be resolved in a second job. Finally, a join between the last job output and `TP5`, over `?Z`, provides the final result. Thus, the query is satisfied with three jobs. Another choice, shown in Figure 5(c), is to run a single job which joins `TP1` and `TP3`, over `?X`, and `TP4` and `TP5`, over `?Z`. Then, a second job makes the multi-way join between the outputs of the first job and `TP2` over, `?Y`. In this case, the query is resolved with only two jobs because, unlike the previous plan, two of the joins can be done in parallel.

Resource popularity is another important issue which must be considered. Urbani et al. [58] point out that some resources are more popular than others in an RDF dataset. Thus, `Reduce` tasks can receive very disparate quantities of data and it can lead to lower performance because the whole process depends on the slower task [25]. For instance, in the query of Figure 5(a), it can be expected that the join between `TP4` and `TP5` ends sooner than the join between `TP1` and `TP3`. Thus, query optimization must take into account the fact that resources are distributed in a biased way.

Finally, **I/O disk operations** are the main bottleneck in MapReduce jobs, as stated before. This fact is especially relevant for semantic data processing because of the schema-relaxed nature of RDF. In practice, this feature means that triples can be located anywhere in the dataset. Thus, how data is effectively encoded is determinative for parsing and processing triples. If this fact is not considered, each MapReduce job can be forced to read the whole dataset [28, 26].

5 CLASSIFICATION

As stated, different MapReduce-based solutions have been proposed for SPARQL resolution on a large scale. Thus, we propose a specific classification which allows us to review these solutions in a coherent way. To do this, we divide the technique according to the complexity that their nodes must assume within the MapReduce cluster. We consider two main families that we refer to as **native** and **hybrid** solutions.

Native solutions. This first family considers all solutions relying exclusively on the MapReduce framework. That is, queries are resolved with a series of MapReduce jobs on low complexity nodes, typically disregarding the help of specific semantic technology in each node of the cluster.

Thus, the overall SPARQL resolution performance mainly depends on (a) physical tuning decisions, and (b) processing strategies which reduce the number of required jobs. As for (a), we consider two main decisions to optimize the physical aspects:

- *Data storage* is one the main bottlenecks within the MapReduce ecosystem. Note that data are persistently read from disk and stored again to communicate `Map` and `Reduce` stages. Thus, how the data are organized for storage purposes in HDFS is one of the considerations addressed by the existing solutions.
- *Data indexing* is an immediate improvement for reducing the aforementioned I/O costs. For this purpose, some approaches use a NoSQL database (in general, HBase) on top of HDFS. These approaches leverage database indexes to improve RDF retrieval within individual nodes, therefore improving TP resolution within the `Map` stage. These random access capabilities allow more complex `Map` and `Reduce` stages to be implemented in practice.

Regardless of this physical tuning, all approaches carry out specific strategies for reducing the number of MapReduce jobs required for general SPARQL resolution. Both physical decisions and these processing strategies are reviewed in Section 6 within the corresponding approaches.

Hybrid solutions. This second family, in contrast to native solutions, deploys MapReduce clusters on more complex nodes, typically installing mono-node state-of-the-art RDF stores. This decision allows each node to partially resolve SPARQL queries on its stored subgraph, so that MapReduce jobs are only required when individual node results must be joined. In this scenario, *triples distribution* is an important decision to minimize (or even avoid) the number of MapReduce jobs required for query resolution.

Table 1 summarizes the most relevant solutions, in the state of the art, according to the above classification. The corresponding papers are reviewed in the next two sections.

In addition to the reviewed papers, it is worth noting that there exist some other proposals [54, 47, 50, 31] which are deployed on top of additional frameworks running over Hadoop, such as Pig [41] or Hive [56, 57]. These solutions use high-level languages over Hadoop to hide `map` and `reduce` task complexities from developers. However, we do not look at them in detail because their main contributions are related to higher level details, while their internal MapReduce configurations respond to the same principles discussed below.

6 NATIVE SOLUTIONS

This section reviews the most relevant techniques within the native solutions. Attending to the previous classification, we analyze solutions running on native HDFS storage (Section 6.1), and NoSQL-based proposals (Section 6.2).

6.1 HDFS-based Storage

Before going into detail, it is worth noting that all these solutions use **single line notations** for serializing RDF data in plain files stored in HDFS. Some solutions use

Table 1: Classification and strategies of MapReduce-based solutions addressing SPARQL resolution.

Solution Type	Purpose	Solution	Reason	Papers
Native solutions	Simplify automated processing	Single line notations	Each triple is stored in a separate line	[28], [26], [27], [39]
	Reduce storage requirements	Substitution of common prefixes by IDs	Data size reduction	[28], [26], [27]
		Division of data in several files by predicate and object type	Only files with corresponding TPs will be read	
	Improve data processing speed	Storage of all triples with the same subject in a single line	Improve reading speed of queries with large number of results	[48]
		Map-side joins	Reduce data shuffled and transferred between tasks	[19], [49]
		NoSQL solutions	Provide indexed access to triples	[55], [43], [49]
	Reduce number of MapReduce jobs	Greedy algorithm	Optimize Star Queries	[28], [26], [39]
		Multiple Selection algorithm	Optimize Path Queries	
		Early elimination heuristic	Prioritize jobs that completely eliminate variables	[27]
		Clique-based heuristic	Resolve queries with map-side joins	[19]
Hybrid solutions	Allow parallel subgraph processing	Graph Partitioning	Each node receives a significant subgraph	[25], [36]
		Parallel subgraph processing	Each node resolves subgraph joins	

straight N-Triples format [20] for storage purposes [39], while others preprocess data and transform them to their own formats. This simple decision simplifies RDF processing, because triples can be individually parsed line-by-line. In contrast, formats like RDF/XML [3] force the whole dataset to be read in order to extract a triple [28, 26, 39, 46, 25].

RDF storage issues are addressed by Rohloff and Schantz [48], Husain et al. [28, 26, 27] and Goasdoué and Kaoudi [19] in order to reduce space requirements and data reading on each job. Rohloff and Schantz [48] transform data from N3 into a plain text representation in which triples with the same subject are stored in a single line. Although it is not an effective approach for query processing (in which a potentially small number of triples must be inspected), it is adequate in the MapReduce context because large triple sets must be scanned to answer less-selective queries [48].

Another immediate approach is based on common **RDF prefix substitution** [28]. In this case, all occurrences of RDF terms (within different triples) are replaced by short IDs which reference them in a dictionary structure. It enables spatial savings, but also parsing time because the amount of read data is substantially reduced.

Husain et al. [26] focus on storage requirements and I/O costs by **dividing data into several files**. This decision allows data to be read in a more efficient way, avoiding the reading of the whole dataset in its entirety. This optimization comprises two sequential steps:

1. A predicate-based partitioning is firstly performed. Thus, triples with the same predicate are stored as pairs `(subject,object)` within the same file.
2. Then, these files are further divided into *predicate-type* chunks in order to store together resources of the same type (e.g. `ex:student` or `ex:degree` in the example in Figure 1). This partitioning is performed in two steps:
   (a) *Explicit type information* is firstly used for partitioning. That is, `(subject,object)` pairs, from the `rdf:type` predicate file, are divided again into smaller files. Each file stores all subjects of a given type, enabling resources of the same type to be stored together.
   (b) *Implicit type information* is then applied. Thus, each predicate file is divided into as

many files as different object types it contains. This division materializes the *predicate-type* split. Note that this is done with the information generated in the previous step. Thus, each chunk contains all the `(subject,object)` pairs for the same predicate and type.

General query resolution performs on an iterative algorithm which looks for the files required for resolving each TP in the query. According to the TP features, the algorithm proceeds as follows:

- If both predicate and object are variables (but the type has been previously identified), the algorithm must process all *predicate-type* files for the retrieved type.
- If both predicate and object are variables (but the type has not been identified), or if only the predicate is variable, then all files must be processed. Thus, the algorithm stops because no savings can be obtained.
- If the predicate is bounded, but the object is variable (but the type has been previously identified), the algorithm only processes the *predicate-type* file for the given predicate and the retrieved type.
- If the predicate is bounded, but the object is variable (but the type has not been identified), all *predicate-type* files must be processed.

For instance, when making split selection for the query of Figure 5(a), the selected chunks would be the following:

- `TP1`: type file `ex:student`.
- `TP2`: type file `ex:degree`.
- `TP3`: predicate-type file `ex:study-ex:degree`.
- `TP4`: predicate-type file `ex:hasProfessor-ex:professor`.
- `TP5`: type file `ex:professor`.

Goasdoué and Kaoudi [19] focus their attention on another MapReduce issue: the job performance greatly depends on the amounts of intermediate data shuffled and transmitted from `Map` to `Reduce` tasks. Their goal is to partition and place data so that most of the joins can be resolved in the Map phase. Their solution replaces the HDFS replication mechanism by a personalized one, where each triple is also replicated three times, but in three different types of partition: subject partitions, property (predicate) partitions, and object partitions. For a given resource, the subject, property, and object partitions of this resource are placed in the same node. In addition, subject and object partitions are grouped within a node by their property values. The property `rdf:type` is highly skewed, hence its partition is broken down into smaller partitions to avoid performance issues. In fact, property values in RDF are highly skewed in general [32], which can cause property partitions to differ greatly in size. This issue is addressed by defining a threshold: when creating a property partition, if the number of triples reaches the threshold, another partition is created for the same property. It is important to note that this replication method improves the performance, but at the cost of fault-tolerance: HDFS standard replication policy ensures that each item of the data is stored in three different nodes, but with this personalized method, this is not necessarily true.

Husain et al. [28] and Myung et al. [39] focus on **reducing the number of MapReduce jobs** required to resolve a query. Both approaches use algorithms to select variables in the optimal way. Their operational foundations are as a follows. In a first step, all TPs are analyzed. If they do not share any variable, no joins are required and the query is completed in a single job. Note that while a cross product would be required in this case, this issue is not addressed by these works. Otherwise, there are TPs with more than one variable. In this case, variables must be ordered and two main algorithms are considered:

- The *greedy algorithm* promotes variables participating in the highest number of joins.
- The *multiple selection algorithm* promotes variables which are not involved in the same TP because these can be resolved in a single job.

Both algorithms can be combined for variable ordering. Whereas Myung et al. [39] make an explicit differentiation between the algorithms, Husain et al. [28] implement a single algorithm which effectively integrates them. These algorithms are simple, report quick performance, and lead to good results. However, they are not always optimal. In a later work, Husain et al. [26] obtain each possible job combination and select the one reporting the lowest estimate cost. This solution, however, is reported as computationally expensive.

Husain et al. [27] revisit their previous work and develop *Bestplan*, a more complex solution for TPs selection. In this solution, jobs are weighted according to their estimated cost. The problem is then defined as a search algorithm in a weighted graph, where each vertex represents a state of TPs involved in the query, and edges represent a job to make the transition from one state to another. The goal is to find the shortest weighted path between the initial state v_0, where each TP is unresolved, to the final state v_{goal}, where every TP is resolved. How-

ever, it is possible (in the worst case) that the complexity of the problem were exponential in the number of joining variables. Only if the number of variables is small enough, is it a feasible solution for generating the graph and finding the shortest path.

For higher numbers of joining variables, the *Relaxed-Bestplan* algorithm is used. It assumes uniform cost for all jobs; *i.e.*, the problem is to find the minimum number of jobs. This concern can also be infeasible, but it is possible to implement a greedy algorithm that finds an upper bound on the maximum number of jobs that performs better than the greedy algorithm defined in [28]. This algorithm is based on an *early elimination heuristic*. That is, jobs that *completely eliminate* variables are selected first, where *complete elimination* means that this variable is resolved in every TP it appears.

On the other hand, Goasdoué and Kaoudi [19] represent the query as a *query graph* where nodes are TPs, and edges model join variables between them (these are labeled with the name of the join variables). Then, the concept *clique subgraph* is proposed from the well-known concept of *clique* in Graph Theory; a clique subgraph G_v is the subset of all nodes which are adjacent to the edge labeled with the variable v (*i.e.: triples that share variable* v). Using this definition, possible queries are divided into the following groups:

- *1-clique query:* where the query graph contains a single clique subgraph. These queries can be resolved in the `Map` stage of a single job, because the join can be computed locally at each node.

- *Central-clique query:* where the query graph contains a single clique subgraph that overlaps with all other clique subgraphs. These queries can be resolved in a single complete job. The query can be decomposed into 1-clique queries that can be resolved in the `Map` phase of a MapReduce job, and then the results of these joins can be joined in the variable of the common clique on the `Reduce` stage.

- *General query*: This is neither 1-clique nor central-clique query. These queries require more than one job to be resolved. A greedy algorithm, referred to as *CliqueSquare*, is used to select join variables. This algorithm decomposes queries into clique subgraphs, evaluates joins on the common variables of each clique, and finally collapses them. All this processing is implemented in a MapReduce job. As cliques are collapsed, each node in the query graph represents a set of TPs.

6.2 NoSQL Solutions

In order to improve RDF retrieval in each node, some approaches use the NoSQL distributed database *HBase* [55, 43, 49] on top of HDFS. These solutions perform triples replication, in diverse tables, and use some indexing strategies to speed up query performance for TPs with distinct unbound variables. The main drawback of these approaches is that each triple is now stored many times, one for each different table, and this spatial overhead is added to the HDFS replication itself (with a default replication factor of three).

Sun and Jin [55] propose a sextuple indexing similar to the one of Hexastore [59] or RDF3X [40]. It consists of six indexes: *S_PO, P_SO, O_SP, PS_O, SO_P*, and *PO_S*, which cover all combinations for unbound variables. Thus, all TPs are resolved with one access to the corresponding index.

Papailiou et al. [43] reduce the number of tables to three, corresponding to indexes *SP_O, OS_P*, and *PO_S*. TPs with only one bound resource are resolved with a range query `[<resource>,increment(<resource>)]`. To further improve query performance, all indexes store only the 8-byte MD5Hashes of {`s,p,o`} values; a table containing the reverse MD5Hash to values is kept and used during object retrieval.

Schätzle and Przyjaciel-Zablocki [49] reduce the number of tables even more, using only two, corresponding to indexes *S_PO* and *O_PS*. The *HBase Filter API* is used for TPs which bound `(subject and object)` or `(object and predicate)`. In turn, predicate bounded TPs can be resolved using the *HBase Filter API* on any of the two tables.

Although these solutions rely on HBase for triple retrieving, join operations are still performed via `Map` and `Reduce` operations. Sun and Jin [55] use similar algorithms to that described previously for [28, 39]. Papailiou et al. [43], on the contrary, develop a complex join strategy where joins are performed differently depending on BGP characteristics. The different join strategies are:

- The **map phase join** is the base case and follows the general steps described in section 4. That is, the mappers read the triples and emit *(key, value)* pairs in which i) the *keys* correspond to the join variable bindings, and ii) the *values* correspond to the bindings for all other variables included in the TPs of the join (if exist) calculated in previous jobs. In turn, the reducers merge, for each key, the corresponding list of values in order to perform the join. Although this strategy is named *map phase join*, the actual join is performed in the `Reduce` phase (the name only refers to when data is extracted from HBase).

- The **reduce phase join** is used when one of the TPs retrieves a very small number of input data compared to the rest. In this case, the `Map` stage is the same as in the *Map phase join*, but only this TP is used as input. It is in the `Reduce` stage where, for each mapped binding, data are retrieved if they match other TPs.

- The **partial input join** is similar to *Reduce phase join*, but allows an arbitrary number of TPs to be selected for extracting their results in the `Map` stage. These selection use information gathered during the bulk data loading to HBase.

- Instead of launching a MapReduce job, the **centralized join** performs the join operation in a single node. This is only an efficient choice when the input data size is small, and the initialization overhead of a MapReduce job is a major factor in query performance.

Furthermore, Schätzle and Przyjaciel-Zablocki [49] develop a join strategy named *Map-Side Index Nested Loop Join* (*MAPSIN join*). This strategy performs the join in the `Map` stage instead of the `Reduce` one. It firstly performs a distributed table scan for the first TP, and retrieves all local results from each machine. For each possible variable binding combination, the `Map` function is invoked for retrieving compatible bindings with the second TP. The computed multiset of solutions is stored in HDFS. This approach highly reduces the network bandwidth usage, as only compatible data for the second TP needs to be transferred to the nodes which run the `Map` tasks. Note that, when the join is materialized in the `Reduce` stage, all possible bindings are transferred from `Map` to `Reduce` tasks. Joins involving three or more TPs are computed successively. For each additional TP, a `Map` stage is performed after joining the previous TPs. In this case, the `Map` function is invoked for each solution obtained in the previous joins. Finally, in the case of multi-way joins, compatible bindings for all TPs are retrieved from HBase in a single `Map` stage. Finally, for queries involving a high selective TP (retrieving few results), the `Map` function is invoked in one machine for avoiding the MapReduce initialization.

7 Hybrid Solutions

The approaches reviewed in the previous section strictly rely on the MapReduce framework for resolving SPARQL queries. Some other techniques introduce specific semantic technology in each node of the cluster. In general, this class of solutions deploy an RDF store in each cluster machine and distribute the whole dataset among each node. The motivation behind this idea is to manage a significant RDF subgraph in each node in order to minimize inter-node communication costs. In fact, when join resolution can be isolated within a single node, the complete query is resolved in parallel using MapReduce. These queries are defined as *Parallelizable Without Communication* (PWOC) by Huang et al. [25]. In this case, the final results are just the addition of each node output (note that a cross product would be required in this case, but this case is not considered in the original paper). If a query is not PWOC, it is decomposed in parallelizable queries and their result is finally merged with MapReduce.

A straightforward, but smart, data distribution performs hashing by subject, object or subject-object (resources which can appear as subject or object in a triple) [36]. This hash-based partitioning is also used in multiple distributed RDF Stores such as YARS2 [23], Virtuoso Cluster [14], Clustered TDB [42], or CumulusRDF [33]. In the current scenario, hash-partitioning enables TPs sharing a variable resource to be resolved without inter-node communication. This result is especially interesting for star-shaped query resolution, but more complex queries require intermediate results to be combined and this degrades the overall performance [25]. Edge-based partitioning is another effective means of graph distribution. In this case, triples which share subject and object are stored in the same node. However, triples describing the same subject can be stored in different nodes, hindering star-shaped query resolution [25]. Finally, Huang et al. [25] and Lee and Liu [36] perform a vertex-based partitioning. By considering that each triple models a graph edge, this approach distributes subsets of closer edges in the same machines. This partitioning involves the following three steps:

1. The whole graph is vertex-based partitioned in disjoint subsets. This class of partitioning is well-known in Graph Theory, so standard solutions such as the *Metis partitioner* [30], can be applied.

2. Then, triples are assigned to partitions.

3. Partitions are finally expanded through controlled triple replication.

It is worth noting that `rdf:type` generates undesirable connections: every resource is at two hops of any other resource of the same type. These connections make the graph more complex and reduce the quality of graph partitioning significantly. Huang et al. [25] remove triples with predicate `rdf:type` (along with triples with similar semantics) before partitioning. Highly-connected vertices are also removed because they can damage quality in a similar way. In this case, a threshold is chosen and

those vertices with more connections are removed before partitioning.

Huang et al. [25] create partitions including triples with the same subject. In turn, Lee and Liu [36] obtain partitions for three different kinds of groups:

- *Subject-based triple groups*, comprising those triples with the same subject.
- *Object-based triple groups*, comprising those triples with the same object.
- *Subject-object-based triple groups*, comprising those triples with the same subject *or* object.

The overall query performance could be improved if triples replication is allowed. It enables larger subgraphs to be managed and queried, yielding configurable space-time tradeoffs [25]. On the one hand, storage requirements increase because some triples may be stored in many machines. On the other hand, query performance improves because more queries can be locally resolved. Note that the performance gap between completely parallel resolved queries and those requiring at least one join is highly significant. Huang et al. [25] introduce two particular definitions to determine the best triple replication choice:

- *Directed n-hop guarantee:* an *n-hop guarantee* partition comprises all vertices which act as objects in triples whose subject is in an *(n-1)-hop guarantee* partition. The *1-hop guarantee* partitions corresponds to the partition created by the vertex partitioning method previously described.
- *Undirected n-hop guarantee:* this is similar to the previous one, but it includes each vertex linked to a vertex in the *(n-1)-hop guarantee* partition (*i.e.* vertices which are subject of a triple having its object on the *(n-1)-hop guarantee*).

Lee and Liu [36] propose comparable definitions:

- *k-hop forward direction-based expansion* is similar to *directed n-hop guarantee*. It adds triples with the subject acting as the object of a triple in the partition.
- *k-hop reverse direction-based expansion* adds triples with an object which appears as the subject of a triple in the partition.
- *k-hop bidirection-based expansion* is similar to *undirected n-hop guarantee*. Thus, it adds triples with a resource playing any resource role for a triple in the partition.

Example. We illustrate these definitions using the RDF excerpt shown in Figure 1. Let us suppose that triples are partitioned by subject, and each one is assigned to a different partition. In this case, the *1-hop guarantee* or any type of *1-hop expansion* would simply obtain the initial partitions without adding any additional triples. This is shown in Figure 6(a).

The 2-hop partitions are obtained by including triples "related" to those in the corresponding 1-hop partition. How this relationship is materialized depends on the type of partition. *Directed 2-hop guarantee* and *2-hop forward direction-based expansion* add triples whose subject is object in any triple within the 1-hop partition. In the current example, the triple `(ex:C1, ex:hasProfessor, ex:P1)` is added to partitions 1 and 2, but the partition 3 is unchanged because there are no more triples with their subject included in the 1-hop partition. The resulting partitions are shown in Figure 6(b).

The *2-hop reverse direction-based expansion*, illustrated in Figure 6(c), add triples whose object is a subject in any triple within the 1-hop partition. For the current example, partitions 1 and 2 remain unchanged whereas the partition 3 adds `(ex:S1, ex:study, ex:C1)` and `(ex:S2, ex:study, ex:C1)`.

The *undirected 2-hop guarantee* and *2-hop bidirection-based expansion* add triples whose subject or object appear, as subject or object, in any triple within the 1-hop partition. In our current example, `(ex:C1, ex:hasProfessor, ex:P1)` is added to partitions 1 and 2 because their subject (`ex:C1`) is already in the partition. In turn, `(ex:S1, ex:study, ex:C1)` and `(ex:S2, ex:study, ex:C1)` are added to the partition 3 because their object (`ex:C1`) is also in the partition. The resulting partitions are illustrated in Figure 6(d).

The subsequent 3 and 4-hop partitions are obtained following the same decisions. It can be tested that, in this example, both *undirected 4-hop guarantee* and *4-hop bidirection-based expansion* include the whole graph. □

This n-hop review leads to an interesting result: in fully connected graphs, both *undirected k-hop guarantee* and *k-hop bidirection-based expansion* partitions will eventually include the whole graph if *n*/*k* is sufficiently increased. However, this is not true for directed guarantees/expansions, as some resources can be connected by the direction which is not considered [25].

To determine if a query is completely resolved in parallel, centrality measures are used. Huang et al. [25] use the concept of *Distance of Farthest Edge* (DoFE). The vertex of a query graph with the smallest DoFE will be considered as the *core*. If the DoFE of the core vertex is less than or equal to the n-hop guarantee, then the query is PWOC. It is worth noting that if directed n-

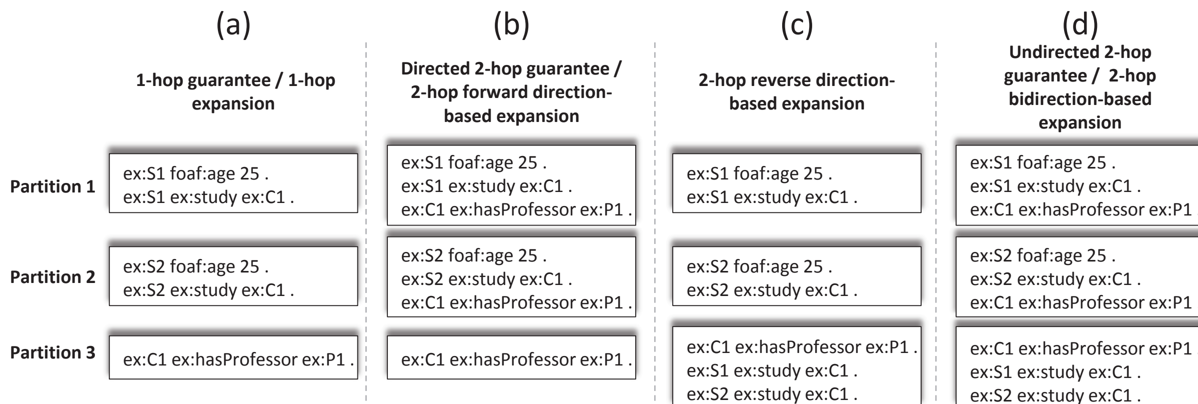

Figure 6: Example of different hop partitions over the RDF graph example in Figure 1.

hop guarantee is used, the distance must be measured considering the query as a directed graph; if undirected n-hop guarantee is used, the query can be considered as an undirected graph. Lee and Liu [36] propose similar definitions with the name of *center vertex* and *radius*. Radius can be calculated as *forward radius* (when using forward direction-based expansion), *reverse radius* (when using reverse direction-based expansion), or *bidirectional radius* (when using bidirectional-based expansion). In these cases, the query graph must be considered as directed, inversely directed, and undirected.

If triples replication is used, it is possible that more than one partition could resolve a query. This could lead to duplicate results when resolving PWOC queries. Huang et al. [25] address this issue in the following way: when a partition is created, additional triples with the form `(`v`,<isOwned>,"Yes")` are added, where v corresponds to core vertexes of the partition (*i.e.* not added as an n-hop guarantee). When resolving a query in parallel, an additional TP with the form `(`*core*`,<isOwned>,"Yes")` is added to the query.

8 Analysis of Results

This section summarizes the experimental results provided by the authors of the most prominent techniques described in the previous sections. It is worth mentioning that any performance comparison would be unfair, as the solutions are tested under different configurations and most of them do not compare to each other. These variations include different node configurations and cluster compositions; the version of Hadoop used in the experiments and also its configuration; and the datasets size. Nevertheless, all of them use the well-known Lehigh University Benchmark[21] (LUBM), obtaining datasets from LUBM(100) to LUBM(30K). This benchmark allows synthetic data, of arbitrary size, to be generated from a university ontology. It also provides a set of 14 queries varying in complexity. Thus, we aim to analyze how solutions face two correlated dimensions: i) **dataset size** and ii) **resolution performance** at incremental query complexity.

8.1 Native solutions on HDFS

As stated, native solutions running on HDFS make use exclusively of MapReduce infrastructure for SPARQL resolution. On the one hand, RDF is stored using different file configurations within HDFS. On the other hand, SPARQL queries are resolved with successive jobs across the nodes. It is worth noting that all techniques analyzed in this section use multi-way joins for query optimization.

The initial work by Husain et al. [28] proposes a promising specific optimization on the basis of organizing triples in files by certain properties. They perform, though, a reduced evaluation with a cluster of 10 nodes, aimed at testing the feasibility and scalability of the proposal. They report runtime for six queries from LUBM on incremental dataset sizes. The solution scales up to $1,100$ million triples and shows sublinear resolution time *w.r.t.* the number of triples. However, no comparisons are made against any other proposal, so its impact within the state of the art cannot be quantified. Their next solution [26] fills this gap and evaluates the approach (again on a cluster of 10 nodes) against mono-node stores: BigOWLIM[6] and Jena[7] (in-memory and the SDB model on disk). The latest solution by Husain et al. [27] compare their approach against the mono-node RDF3X [40]. In this latter comparison, larger datasets are tested, ranging from LUBM(10K), with 1.1 billion triples, to LUBM (30K), with 3.3 billion triples. In addition to LUBM,

[6]http://www.ontotext.com/owlim/editions
[7]http://jena.apache.org/

a subset of the SP^{2}Bench Performance Benchmark [52] is also used as evaluation queries. These last works reach similar conclusions. As expected, the Jena in-memory model is the fastest choice for simple queries, but it performs poorly at complex ones. Moreover, it runs out of memory on a large scale (more than 100 million triples). Jena SDB works with huge sizes, but it is one order of magnitude slower than HadoopRDF. In general, BigOWLIM is slightly slower in most dataset sizes, and slightly faster in the 1 billion dataset (mostly because of its optimizations and triple pre-fetch). A detailed review shows that BigOWLIM outperforms the MapReduce proposal in simple queries (such as Q12), whereas it is clearly slower in the complex ones (*e.g.* Q2 or Q9). They also evaluate the impact of the number of reducers, showing no significant improvement in the performance with more than 4 reducers. RDF3X performs better for queries with high selectivity and bound objects (*e.g.* Q1), but HadoopRDF outperforms RDF3X for queries with unbound objects, low selectivity, or joins on large amounts of data. Moreover, RDF3X simply cannot execute the two queries with unbound objects (Q2 and Q9) with the LUBM(30K) dataset.

Goasdoué and Kaoudi [19] compare their solution with respect to HadoopRDF [27]. Datasets LUBM(10K) and LUBM(20K) are used in the tests. All the queries correspond to 1-clique or central-clique queries, and thus they can be resolved in a single MapReduce job. This allows CliqueSquare to outperforms HadoopRDF in each query by a factor from 28 to 59.

Myung et al. [39] focus their evaluation on testing their scalability with LUBM at incremental sizes. However, the maximum size is only LUBM(100), *i.e.* synthetic-generated triples from 100 universities. In contrast, Husain et al. [28] start their evaluation from 1,000 universities. Therefore, the very limited experimentation framework prevents the extraction of important conclusions. Nonetheless, they also verify the sublinear performance growth of the proposal (*w.r.t.* the input) and the significant improvement using multi-way joins versus two-way joins.

8.2 Native solutions on NoSQL

As explained before, this class of solutions replaces the plain HDFS storage by a NoSQL database in order to improve the overall data retrieval performance. The following results support this assumption, showing interesting improvements for query resolution.

Sun and Jin [55] make an evaluation using LUBM [21] datasets from 20 to 100 universities. Although no comparisons are made with respect to any other solutions, their results report better performance for growing dataset sizes. This result is due to the impact of MapReduce initialization decreases for larger datasets.

Papailiou et al. [43] compare themselves with the mono-node RDF3X and with the MapReduce-based solution HadoopRDF [27]. The experiments comprise a variable number of nodes for the clusters and a single machine for RDF3X. This machine deploys an identical configuration to that used for the nodes in the clusters. LUBM datasets are generated for 10,000 and 20,000 universities, comprising 1.3 and 2.7 billion triples respectively. Their solution shows the best performance for large and non-selective queries (Q2 and Q9), and outperforms HadoopRDF by far for centralized joins. Nonetheless, it is slightly slower than RDF3X for this case. Regarding scalability, execution times are almost linear *w.r.t.* the input when the number of nodes does not vary within the node, and decreases almost linearly when more nodes are added to the cluster.

Schätzle and Przyjaciel-Zablocki [49] perform an evaluation of their MAPSIN join technique over a cluster with 10 nodes. They also use the SP^{2}Bench in addition to LUBM. For LUBM, datasets from 1,000 to 3,000 universities are generated; for SP^{2}Bench, datasets from 200 million to 1 billion triples are generated. Their results are compared against PigSPARQL [50], another work from some of the same authors that uses Pig to query RDF datasets. Both approaches scale linearly, but MAPSIN on HBase enable an efficient way of Map-Side join because it reduces the necessary data shuffle phase. This allows join times to be reduced from 1.4 to 28 times with respect to the compared technique.

8.3 Hybrid solutions

The hybrid solutions aim to minimize the number of MapReduce jobs, resolving queries in local nodes and restricting the communication and coordination between nodes just for complex queries (cross-joins between nodes). This is only effective on the basis of a previous smart subgraph partitioning.

Huang et al. [25] establish a fixed dataset of 2,000 universities (around 270 million triples) for their evaluation, and do not compare incremental sizes. They perform on a cluster of 20 nodes, and their proposal is built with the RDF3X [40] triple store working in each single node. First, they compare the performance of RDF3X on a single node against the SHARD [48] native solution, showing that this latter is clearly slower because most joins require a costly complete redistribution of data (stored in plain files). In contrast, subject-subject joins can be efficiently resolved thanks to the hash partitioning. Next, the performance of Huang et al.'s solution is evaluated against RDF3X on a single node. Once again, the simplest queries (Q1, Q3, Q4, etc.) run faster on a single

machine, whereas the hybrid MapReduce solution dramatically improves the performance of complex queries (Q2, Q6, Q9, Q13 and Q14), ranging from 5 to 500 times faster. The large improvement is achieved for large clusters, because chunks are small enough to fit into main memory. In addition, they verify that the 1-hop guarantee is sufficient for most queries, except those with a larger diameter (Q2, Q8 and Q9), in which the 2-hop guarantee achieves the best performance and, in general, supports most SPARQL queries (given the small diameter of the path queries).

Finally, Lee and Liu [36] yield to a very similar approach. They also install RDF3X in each single node (20 nodes in the cluster), but their performance is only compared against a single-node configuration. In contrast, they perform on incremental sizes (up to 1 billion triples) and study different benchmarks besides LUBM. They also conclude that a 2-hop guarantee is sufficient for all queries (it leads to similar results to even a 4-hop guarantee) and, in each case, this subgraph partitioning is more efficient than the hash-based data distribution used, for instance, in SHARD [48]. The single-node configuration does not scale on most datasets, whereas the scalability of the MapReduce system is assured once the resolution time increases only slightly at incremental dataset sizes.

9 Discussion and Conclusions

MapReduce is designed to process data in distributed scenarios under the assumption of no intercommunication between `Map` and `Reduce` tasks during their execution. However, RDF data is interweaved because of its graph nature, and triple relationships are spatially arbitrary. For these reasons, multiple MapReduce jobs can be necessary to resolve SPARQL queries. Moreover, a plain data storage and organization overloads the processing, and expensive costs must be paid whenever a new job starts. Thus, efficient SPARQL resolution on MapReduce-based solutions is mainly based on optimizing RDF data management and minimizing the number of MapReduce jobs required for query resolution.

We review the most relevant proposals throughout the paper, establishing a categorization in two different groups: (i) native solutions and (ii) hybrid solutions. Native solutions resolve SPARQL queries using MapReduce tasks exclusively, whereas hybrid solutions perform subgraph resolution in each node, and resort to MapReduce to join the results of each subgraph. In native solutions, the main contributions relate to reducing the number of jobs needed to perform joins, and to data organization. Data can be stored in HDFS, where data must be organized in files, or in another solution such as HBase, where triples can be indexed for faster access. In hybrid solutions, the main contributions are related to how data is partitioned in order to obtain optimal subgraphs.

Although many of the prominent solutions cannot be directly compared, given their different configurations, a detailed analysis of their results draws significant conclusions: (i) MapReduce-based solutions scale almost linearly with respect to incremental data sizes, (ii) they perform worse than classical mono-node solutions with simple queries or small datasets, but (iii) they outperform these solutions when the query complexity or the dataset size increases.

The state-of-the-art approaches also evidence that data must be preprocessed (i) to obtain easily readable notation, (ii) to enable partial reads to be done, and (iii) to reduce storage requirements. In addition, two of the reviewed papers also organize data in such a way that the process can capitalize on data locality and perform joins on `Map` tasks [49, 19]. It highly reduces data shuffling and improves performance. Although this preprocessing step could be computationally expensive, it is a once-only task which improves performance dramatically. In this scenario, binary RDF serialization formats such as RDF/HDT [16] could enhance the overall space/time tradeoffs. Note that these approaches can manage RDF in compressed space, enabling TP resolution at high levels of the memory hierarchy.

Apparently, the more complex the solutions, the better the performance results, but this comes at an important cost. On the one hand, they incur serious storage overheads because of data redundancy: NoSQL solutions can require up to 6 times the space of native solutions, whereas hybrid solutions report up to 4.5 times just for 2-hop partitions. On the other hand, the simpler native solutions are easier to implement in vanilla MapReduce clusters, which make deployment in shared infrastructures or in third party services (such as AWS Elastic MapReduce[8]) an almost straightforward operation. As complexity grows, solutions are harder to implement.

While these works showcase relevant contributions for SPARQL resolution using MapReduce, the absence of communication between tasks continues to present an important challenge when joins are involved. This can be seen as a general MapReduce issue that motivates different researches. Some proposals add an additional phase to the MapReduce cycle. For instance, Map-Reduce-Merge [61] adds an additional function at the end of the MapReduce cycle in order to support relational algebra primitives without sacrificing its existing generality and simplicity. In turn, Map-Join-Reduce [29] intro-

[8] `http://aws.amazon.com/elasticmapreduce`

duces a *filtering-join-aggregation* programming model which is an extension of the MapReduce programming model. Tuple MapReduce [17], though, takes a different approach and proposes a theoretical model that extends MapReduce to improve parallel data processing tasks using compound-records, optional in-reduce ordering, or intersource datatype joins. In addition, there are specific proposals for providing support to iterative programs like Twister [13] or HaLoop [7]. This aims to improve data locality for those tasks accessing to the same data (even in different jobs), while providing some kind of caching of invariant data. Thus, it is expected that all these general-purpose proposals will feedback specific applications, and SPARQL resolution on MapReduce will be benefited with advances from these lines of research.

ACKNOWLEDGMENTS

The authors would like to thank Mercedes Martínez-González for her suggestions and feedback on the initial development of this paper.

REFERENCES

[1] F. Afrati and J. Ullman, "Optimizing Joins in a Map-reduce Environment," in *Proc. of the 13th International Conference on Extending Database Technology (EDBT)*, 2010, pp. 99–110.

[2] R. Anglés and C. Gutiérrez, "The Expressive Power of SPARQL," in *Proc. of the International Semantic Web Conference (ISWC)*, 2008, pp. 114–129.

[3] D. Beckett, Ed., *RDF/XML Syntax Specification (Revised)*, ser. W3C Recommendation, 2004, http://www.w3.org/TR/rdf-syntax-grammar/.

[4] T. Berners-Lee, R. Fielding, and L. Masinter, "RFC 3986, Uniform Resource Identifier (URI): Generic Syntax," 2005.

[5] C. Bizer, T. Heath, and T. Berners-Lee, "Linked Data - The Story So Far," *International Journal on Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, 2009.

[6] D. Borthakur, "HDFS Architecture Guide," 2008, available at: http://hadoop.apache.org/docs/stable1/hdfs_design.html.

[7] Y. Bu, B. Howe, M. Balazinska, and M. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[8] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computing Systems*, vol. 26, no. 2, article 2, 2008.

[9] R. Cyganiak, "A relational algebra for SPARQL," HP Laboratories Bristol, Tech. Rep. HPL-2005-170, 2012, available at: http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html.

[10] C. David, C. Olivier, and B. Guillaume, "A Survey of RDF Storage Approaches," *ARIMA Journal*, vol. 15, pp. 11–35, 2012.

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Largee Clusters," in *Proc. of the 6th Symposium on Operating Systems Design & Implementation (OSDI)*, 2004, pp. 137–150.

[12] M. Duerst and M. Suignard, "RFC 3987, Internationalized Resource Identifiers (IRIs)," 2005.

[13] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-h. Bae, J. Qiu, and G. Fox, "Twister: a Runtime for Iterative MapReduce," in *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010, pp. 810–818.

[14] O. Erling and I. Mikhailov, "Towards web scale RDF," in *Proc. 4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2008.

[15] J. D. Fernández, M. Arias, M. A. Martínez-Prieto, and C. Gutiérrez, *Management of Big Semantic Data*. Taylor and Francis/CRC, 2013, ch. 4.

[16] J. Fernández, M. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary RDF Representation for Publication and Exchange (HDT)," *Journal of Web Semantics*, vol. 19, pp. 22–41, 2013.

[17] P. Ferrera, I. de Prado, E. Palacios, J. Fernandez-Marquez, and G. Di Marzo, "Tuple MapReduce: Beyond Classic MapReduce," in *Proc. of the 12th International Conference on Data Mining (ICDM)*, 2012, pp. 260–269.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 29–43.

[19] F. Goasdoué and Z. Kaoudi, "CliqueSquare: efficient Hadoop-based RDF query processing," *Journes de Bases de Donnes Avances*, pp. 1–28, 2013, available at: http://hal.archives-ouvertes.fr/hal-00867728/.

[20] J. Grant and D. Beckett, Eds., *RDF Test Cases*. W3C Recommendation, http://www.w3.org/TR/rdf-testcases/.

[21] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Journal of Web Semantics*, vol. 3, no. 2, pp. 158–182, 2005.

[22] C. Gutiérrez, C. Hurtado, A. O. Mendelzon, and J. Pérez, "Foundations of Semantic Web

Databases," *Journal of Computer and System Sciences*, vol. 77, pp. 520–541, 2011.

[23] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "Yars2: a Federated Repository for Querying Graph Structured Data from the Web," 2007, pp. 211–224.

[24] M. Henderson, "Multi-Way Hash Join Effectiveness," Ph.D. dissertation, University of British Columbia, 2013.

[25] J. Huang, D. Abadi, and K. Ren, "Scalable SPARQL Querying of Large RDF Graphs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1123–1134, 2011.

[26] M. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham, "Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools," 2010, pp. 1–10.

[27] M. Husain, J. McGlothlin, M. Masud, L. Khan, and B. Thuraisingham, "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1312–1327, 2011.

[28] M. Husain, P. Doshi, L. Khan, and B. Thuraisingham, "Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce," in *Proc. of the 1st International Conference CloudCom*, 2009, pp. 680–686.

[29] D. Jiang, A. Tung, and G. Chen, "Map-Join-Reduce: Toward Scalable and Efficient Data Analysis on Large Clusters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1299–1311, 2011.

[30] G. Karypis and V. Kumar, "Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0," Department of Computer Science, University of Minnesota, Tech. Rep., 1995.

[31] H. Kim, P. Ravindra, and K. Anyanwu, "Optimizing RDF(S) Queries on Cloud Platforms," in *Proc. of the 22nd International World Wide Web Conference (WWW)*, 2013, pp. 261–264.

[32] S. Kotoulas, E. Oren, and F. Van Harmelen, "Mind the Data Skew: Distributed Inferencing by Speeddating in Elastic Regions," in *Proc. of the 19th International Conference on World Wide Web (WWW)*, 2010, pp. 531–540.

[33] G. Ladwig and A. Harth, "CumulusRDF: Linked Data Management on Nested Key-Value Stores," 2011, pp. 30–42.

[34] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy, "The Unified Logging Infrastructure for Data Analytics at Twitter," *Proceedings of the VLDB Endowment*, pp. 1771–1780, 2012.

[35] K.-h. Lee, Y.-j. Lee, H. Choi, Y. Chung, and B. Moon, "Parallel Data Processing with MapReduce: a Survey," *ACM SIGMOD Record*, vol. 40, no. 4, pp. 11–20, 2012.

[36] K. Lee and L. Liu, "Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1894–1905, 2013.

[37] F. Manola and E. Miller, Eds., *RDF Primer*. W3C Recommendation, 2004, `http://www.w3.org/TR/rdf-primer/`.

[38] M. A. Martínez-Prieto, M. Arias, and J. D. Fernández, "Exchange and Consumption of Huge RDF Data," in *Proc. of the 9th Extended Semantic Web Conference (ESWC)*, 2012, pp. 437–452.

[39] J. Myung, J. Yeon, and S.-G. Lee, in *Proc. of the Workshop on Massive Data Analytics on the Cloud (MDAC)*, ser. MDAC '10, 2010, pp. 1–6.

[40] T. Neumann and G. Weikum, "The RDF-3X Engine for Scalable Management of RDF data," *The VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.

[41] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: a not-so-foreign Language for Data Processing," in *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008, pp. 1099–1110.

[42] A. Owens, A. Seaborne, and N. Gibbins, "Clustered TDB: A clustered Triple Store for Jena," ePrints Soton, 2008, available at: `http://eprints.soton.ac.uk/2669740/`.

[43] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris, "H2RDF: Adaptive Query Processing on RDF Data in the Cloud," pp. 397–400, 2012.

[44] J. Pérez, M. Arenas, and C. Gutiérrez, "Semantics and Complexity of SPARQL," *ACM Transactions on Database Systems*, vol. 34, no. 3, pp. 1–45, 2009.

[45] E. Prud'hommeaux and A. Seaborne, *SPARQL Query Language for RDF*, ser. W3C Recommendation, 2008, `http://www.w3.org/TR/rdf-sparql-query/`.

[46] P. Ravindra, V. Deshpande, and K. Anyanwu, "Towards scalable RDF graph analytics on MapReduce," in *Proc. of the Workshop on Massive Data Analytics on the Cloud (MDAC)*, 2010, pp. 1–6.

[47] P. Ravindra, H. Kim, and K. Anyanwu, "An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce," pp. 46–61, 2011.

[48] K. Rohloff and R. Schantz, "High-Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: The Shard Triple-store," in *Proc. of the Programming Support Innovations for Emerging Distributed Applications (PSI EtA)*, 2010, p. 4.

[49] A. Schätzle and M. Przyjaciel-Zablocki, "Cascading Map-Side Joins over HBase for Scalable Join Processing," arXiv:1206.6293, 2012, available at: http://arxiv.org/abs/1206.6293.

[50] A. Schätzle, "PigSPARQL: mapping SPARQL to Pig Latin," in *Proc. of the International Workshop on Semantic Web Information Management (SWIM)*, 2011, p. article 4.

[51] M. Schmidt, M. Meier, and G. Lausen, "Foundations of SPARQL Query Optimization," in *Proc. of the 13th International Conference on Database Theory (ICDT)*, 2010, pp. 4–33.

[52] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "Spˆ 2bench: a sparql performance benchmark," in *Proc. of the 25th International Conference on Data Engineering (ICDE)*, 2009, pp. 222–233.

[53] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proc. of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[54] R. Sridhar, P. Ravindra, and K. Anyanwu, "RAPID: Enabling Scalable Ad-Hoc Analytics on the Semantic Web," in *Proc. of the 8th International Semantic Web Conference (ISWC)*, 2009, pp. 703–718.

[55] J. Sun and Q. Jin, "Scalable RDF Store Based on HBase and MapReduce," in *Proc. of the 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, 2010, pp. 633–636.

[56] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a Warehousing Solution over a Map-Reduce Framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[57] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive-a petabyte scale data warehouse using hadoop," in *Proc. of the 26th International Conference on Data Engineering (ICDE)*, 2010, pp. 996–1005.

[58] J. Urbani, J. Maassen, and H. Bal, "Massive Semantic Web data compression with MapReduce," *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, pp. 795–802, 2010.

[59] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple indexing for semantic web data management," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1008–1019, 2008.

[60] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, 2012.

[61] H.-c. Yang, A. Dasdan, R.-l. Hsiao, and D. Parker, "Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters," in *Proce. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007, pp. 1029–1040.

AUTHOR BIOGRAPHIES

José M. Giménez-García is an ITC Research Master student at the University of Valladolid, Spain. His main interests are related to scalable distributed systems and the Web of Data.

Javier D. Fernández holds a joint PhD in Computer Science from the University of Valladolid, Spain, and the University of Chile, Chile. He is co-author of the HDT W3C Member Submission and his main interests include scalable representations and indexes for querying the Web of Data, data compression and efficient management of Big (semantic) Data.

Miguel A. Martínez-Prieto is Assistant Professor in the Department of Computer Science at the University of Valladolid, Spain. He completed his Ph.D in Computer Science from the same University in 2010. His current research interests are mainly related to data compression and its application to the efficient representation, management, and querying of huge datasets. He is co-author of HDT, the binary format acknowledged by the W3C for publication and exchange of huge RDF in the Web of Data.