algorithms

MDPI

Article

An Effective and Efficient MapReduce Algorithm for Computing BFS-Based Traversals of Large-Scale RDF Graphs

Alfredo Cuzzocrea [1,*], Mirel Cosulschi [2] and Roberto de Virgilio [3]

Received: 2 September 2015; Accepted: 4 January 2016; Published: 11 January 2016
Academic Editor: Bruno Carpentieri

1 DIA Department, University of Trieste and ICAR-CNR, Trieste 34127, Italy; alfredo.cuzzocrea@dia.units.it
2 Department of Computer Science, University of Craiova, Craiova 200585, Romania; mirelc@central.ucv.ro
3 Dipartimento di Informatica e Automazione, Universitá Roma Tre, Rome 00146, Italy; dvr@dia.uniroma3.it
* Correspondence: alfredo.cuzzocrea@dia.units.it; Tel.: +39-040-558-3580; Fax: +39-040-558-3419

Abstract: Nowadays, a leading instance of *big data* is represented by *Web data* that lead to the definition of so-called *big Web data*. Indeed, extending beyond to a large number of critical applications (e.g., *Web advertisement*), these data expose several characteristics that clearly adhere to the well-known *3V properties* (*i.e.*, *volume*, *velocity*, *variety*). *Resource Description Framework* (RDF) is a significant formalism and language for the so-called *Semantic Web*, due to the fact that a very wide family of *Web entities* can be naturally modeled in a *graph-shaped manner*. In this context, *RDF graphs* play a first-class role, because they are widely used in the context of modern Web applications and systems, including the emerging context of *social networks*. When RDF graphs are defined on top of big (Web) data, they lead to the so-called *large-scale RDF graphs*, which reasonably populate the next-generation Semantic Web. In order to process such kind of big data, *MapReduce*, an open source computational framework specifically tailored to *big data processing*, has emerged during the last years as the reference implementation for this critical setting. In line with this trend, in this paper, we present *an approach for efficiently implementing traversals of large-scale RDF graphs over MapReduce* that is based on the *Breadth First Search* (BFS) strategy for visiting (RDF) graphs to be decomposed and processed according to the MapReduce framework. We demonstrate how such implementation speeds-up the analysis of RDF graphs with respect to competitor approaches. Experimental results clearly support our contributions.

Keywords: MapReduce algorithms; BFS-traversals of RDF graphs; effective and efficient algorithms for big data processing

1. Introduction

Big data describe data sets that grow so large that they become unpractical to be processed by traditional tools like database management systems, content management systems, advanced statistical analysis software, and so forth. The reason why they came into the attention of the research community is that the infrastructure to handle these data sets has become more affordable due to *Cloud Computing* and *MapReduce* [1] based open-source frameworks. The vision of everyday people inspecting large amounts of data, easily interacting with graphical interfaces of software applications that handle *data mining tasks*, has indeed enthused final users. On the other hand, those tools have to leverage people's ability to answer analytical processes by inference through large amounts of data.

The problem with this data is not the complexity of the processes involved, but rather the quantity of data and the rate of collecting new data. The following two examples are slightly illustrative of this phenomenon.

- In April 2009, *EBay* was using two Data Warehouses, namely *Teradata* and *Greenplum*, the former having 6.5 PB of data in form of 17 trillion records already stored, and collecting more than 150 billion new records/day, leading to an ingest rate well over 50 TB/day [2].
- In May 2009, *Facebook* estimates an amount of 2.5 PB of user data, with more than 15 TB of new data per day [3].

As another matter of question, the presence of *graph-based data structures* occurs almost everywhere starting with *social networks* like *Facebook* [4], *MySpace* [5] and *NetFlix* [6], and ending with *transportation routes* or *Internet infrastructure architectures*. These graphs keep growing in size and there are several new things that people would like to infer from the many facets of information stored in these data structures. Graphs can have billions of vertices and edges. Citation graphs, affiliation graphs, instant messenger graphs, phone call graphs and so forth, are part of the so-called *social network analysis and mining* initiative. Some recent works show that, despite the previously supposed sparse nature of those graphs, their density increases over time [7,8]. All considering, these trends tend towards the so-called *big Web data* (e.g., [9,10]), the transposition of big data on the Web.

On the other hand, *Semantic Web technologies* allow us to effectively support analysis of large-scale distributed data sets by means of the well-known *Resource Description Framework* (RDF) model [11], which, obviously represents data via a *graph-shaped approach*. As a consequence, in the Semantic Web initiative, the most prominent kind of graph data are represented by *RDF graphs*. This opens the door to a wide family of database-like applications, such as *keyword search over RDF data* (e.g., [12]).

While RDF graphs offer a meaningful data model for successfully representing, analyzing and mining large-scale graph data, the problem of *efficiently processing RDF graphs* is still an open issue in database research (e.g., [13–15]). Following this main consideration, in this paper, we focus on *the problem of computing traversals of RDF graphs*, being computing traversals of such graphs very relevant in this area, e.g., for RDF graph analysis and mining over large-scale data sets. Our approach adopts the *Breadth First Search* (BFS) strategy for visiting (RDF) graphs to be decomposed and processed according to the MapReduce framework. This with the goal of taking advantages from the powerful run-time support offered by MapReduce, hence reducing the computational overheads due to manage large RDF graphs efficiently. This approach is conceptually-sound, as similar initiatives have been adopted in the vest of viable solutions to *the problem of managing large-scale sensor-network data over distributed environments* (e.g., [16,17]), like for the case of Cloud infrastructures and their integration with sensor networks. We also provide experimental results that clearly confirm the benefits coming from the proposed approach.

The remaining part of this paper is organized as follows. In Section 2, we provide an overview of *Hadoop* [18], the software platform that incorporates MapReduce for supporting complex *e*-science applications over large-scale distributed repositories. Section 3 describes in more details principles and paradigms of MapReduce. In Section 4, we provide definitions, concepts and examples on RDF graphs. Section 5 contains an overview on state-of-the-art approaches that represent the knowledge basis of our research. In Section 6, we provide our efficient algorithms for implementing BFS-based traversals of RDF graphs over MapReduce. Section 7 contains the result of our experimental campaign according to which we assessed our proposed algorithm. Finally, in Section 8, we provide conclusions of our research and depict possible future work in this scientific field.

2. Hadoop: An Overview

Hadoop is an open-source project from *Apache Software Foundation*. Its core consists of the MapReduce programming model implementation. This platform was created for solving the problem of processing very large amounts of data, a mixture of complex and structured data. It is used with predilection in the situation where data-analytics-based applications require a large number of computations have to be executed. Similar motivations can be found in related research efforts focused on the issue of effectively and efficiently managing large-scale data sets over *Grid environments* (e.g., [19–21]).

The key characteristic of Hadoop is represented by the property of enabling the execution of applications on thousands of nodes and peta-bytes of data. A typical enterprise configuration is composed by tens or thousands of physical or virtual machines interconnected through a fast network. The machines run a POSIX compliant operating system and a Java virtual machine.

Cutting was the creator of the Hadoop framework, and two papers published by Google's researchers had decisive influence on his work: *The Google File System* [22] and *MapReduce: Simplified Data Processing on Large Clusters* [1]. In 2005, he started to use an implementation of MapReduce in *Nutch* [23], an application software for web searching, and very soon the project becomes independent from *Nutch* under the codename *Hadoop*, with the aim of offering a framework for running many jobs within a cluster.

The language used for the development of the Hadoop framework was Java. The framework is providing the users with a Java API for developing MapReduce applications. In this way, Hadoop becomes very popular in the community of scientists whose name is linked to the big data processing. *Yahoo!* is an important player in this domain, this company having for a longer period a major role in the design and implementation of the framework. Another big player is Amazon [24], company currently offering a web service [25] based on Hadoop that is using the infrastructure provided by the *Elastic Compute Cloud—EC2* [26]. Meanwhile, Hadoop becomes very highly-popular, and, in order to sustain this allegation, it is sufficient just to mention the presence of other important players like *IBM*, *Facebook*, *The New York Times*, *Netflix* [27] and *Hulu* [28] in the list of companies deploying Hadoop-based applications.

The Hadoop framework hides the details of processing jobs, leaving the developers liberty to concentrate on the application logic. The framework has the following characteristic features:

(1) *accessible*—it can be executed on large clusters or on Cloud computing services;
(2) *robust*—it was designed to run on commodity hardware; a major aspect considered during its development was the tolerance to frequent hardware failures and crash recovery;
(3) *scalable*—it scales easily for processing increased amount of data by transparently adding more nodes to the cluster;
(4) *simple*—it allows developers to write specific code in a short time.

There is quite a large application domain for Hadoop. Some examples include online *e*-commerce applications for providing better recommendations to clients looking after certain products, or finance applications for risk analysis and evaluation with sophisticated computational models whose results cannot be stored in a database.

Basically, Hadoop can be executed on each major OS: *Linux*, *Unix*, *Windows*, *Mac OS*. The platform officially supported for production is *Linux* only.

The framework is implementing a master-slave architecture used both for distributed data storage and for distributed computing. Hadoop framework running on a completely-configured cluster is composed of a set of *daemons* or *internal modules* executed on various machines of the cluster. These daemons have specific tasks, some of them running only on a single machine, others having instances running on several machines.

A sample Hadoop cluster architecture is depicted in Figure 1. For a small cluster, *Secondary NameNode* daemon can be hosted on a slave node, meanwhile, on a large cluster, it is customary to separate the *NameNode* and *JobTracker* daemons on different machines. In Figure 1, a master node is running *NameNode* and *JobTracker* daemons and an independent node is hosting the *Secondary NameNode* daemon, just in case of failure in the master node. Each slave machine hosts two daemons, *DataNode* and *TaskTracker*, in order to run computational tasks on the same node where input data are stored.

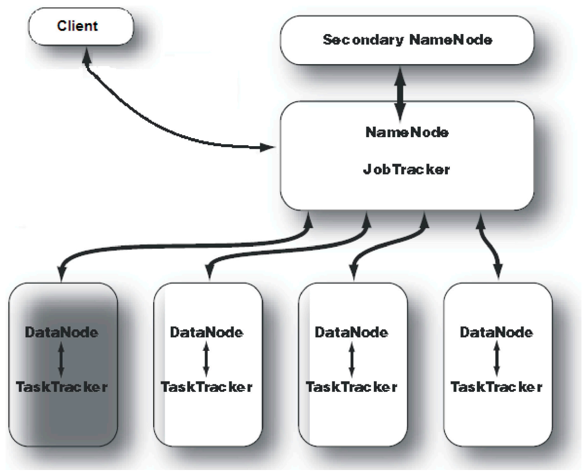


Figure 1. Hadoop cluster architecture.

As regards the proper data storage solution, Hadoop's ecosystem incorporates *HBase* [29]. HBase is a *distributed column-oriented database* (e.g., [30]), which also falls in the category of *NoSQL storage systems* (e.g., [31]), that founds on the underlying *Hadoop Distributed File System* (HDFS) (e.g., [32]) for common data processing routines. HBase has been proved to be highly scalable. HBase makes use of a specific model that is similar to *Google's BigTable* (e.g., [33]), and it is designed to provide fast random access to huge amounts of structured data. As regards the main software architecture organization, HBase performs read/write (fast) routines on HDFS while data consumer and data producer applications interact with (big) data through HBase directly. Recently, due to its relevance, several studies have focused on HBase.

3. MapReduce: Principles of Distributed Programming over Large-Scale Data Repositories

MapReduce is both a programming model for expressing distributed computations on massive amounts of data and an execution framework for large-scale data processing on clusters of commodity servers [1,34]. MapReduce's programming model was released to the computer science community in 2004, being described in [1] after Google was already using this technology for a couple of years. MapReduce's processing scheme soon became very popular, including a broad adoption due to its open-source implementation. A job from MapReduce programming model splits the input data set into several independent blocks that are processed in parallel.

There are two seminal ideas that are encountered in MapReduce, which we report in the following.

- There is a previous well-known strategy in computer science for solving problems: *divide-et-impera*. According to this paradigm, when we have to solve a problem whose general solution is unknown, then the problem is decomposed in *smaller sub-problems*, the same strategy is applied for it, and the results/partial solutions corresponding to each sub-problem into a general solution are combined. The decomposing process continues until the size of the sub-problem is small enough to be handled individually.
- The basic data structures are *key-value pairs* and *lists*. The design of MapReduce algorithms involves usage of *key-value* pairs over arbitrary data sets.

Map and *Reduce* concepts were inspired from *functional programming*. *Lisp*, in particular, introduces two functions with similar meaning, *map* and *fold*. Functional operations have the property of preserving data structures such that the original data remains unmodified and new data structures are created. The *map* function takes as arguments a function f and a list, and it applies the function f to each element of the input list, resulting in a new output list. The *fold* function has an extra argument compared to the *map*: beyond the function f and the input list, there is an accumulator. The function f

is applied to each element of the input list, the result is added to previous accumulator, thus obtaining the accumulator value for the next step.

Although the idea seems to be very simple, implementation details of a *divide-and-conquer* approach are not. There are several aspects with which practitioners must deal. Some are the following ones:

- decomposition of the problem and allocation of sub-problems to workers;
- synchronization among workers;
- communication of the partial results and their merge for computing local solutions;
- management of software/hardware failures;
- management of workers' failures.

The MapReduce framework hides the peculiarities of these details, thus allowing developers to concentrate on the main aspects for implementation of target algorithms.

Technically-wise, for each job, the programmer has to work with the following functions: *mapper*, *reducer*, *partitioner* and *combiner*. The *mapper* and *reducer* have the following signatures:

$$\begin{aligned} Map &: (k_1, v_1) \rightarrow [(k_2, v_2)] \\ Reduce &: (k_2, [v_2]) \rightarrow [(k_3, v_3)] \end{aligned}$$

where [...] denotes a list.

The input of a MapReduce's job is fetched from the data stored in the target distributed file system. For each task of type *map*, a *mapper* is created. This is a method which, applied to each input *key-value* pair, returns a number of intermediary *key-value* pairs. The *reducer* is another method applied to a list of values associated with the same intermediary key in order to obtain a list of *key-value* pairs. Transparently to the programmer, the framework implicitly groups the values associated with intermediary keys between the mapper and the reducer.

The intermediary data reach the *reducer* ordered by key, and there is no guarantee regarding the key order among different reducers. The *key-value* pairs from each *reducer* are stored in the distributed file system. The output values from each *reducer* are stored also in *m* files from the distributed file system, considered these latter as output files for each *reducer*. Finally, the `map()` and `reduce()` functions run in parallel, such that the first `reduce()` call cannot start before the last `map()` function ends. This is an important separation point among different phases. Simplifying, at least the MapReduce developer has to implement interfaces of the two following functions:

$$public\ void\ \mathbf{map}\,(K_1\ key,\ V_1\ value,\ OutputCollector < K_2, V_2 >\ output,\ Reporter\ reporter)\\ throws\ IOException$$

$$public\ void\ \mathbf{reduce}(K_2\ key,\ Iterator < V_2 >\ values,\ OutputCollector < K_3, V_3 >\ output,\\ Reporter\ reporter)\ throws\ IOException$$

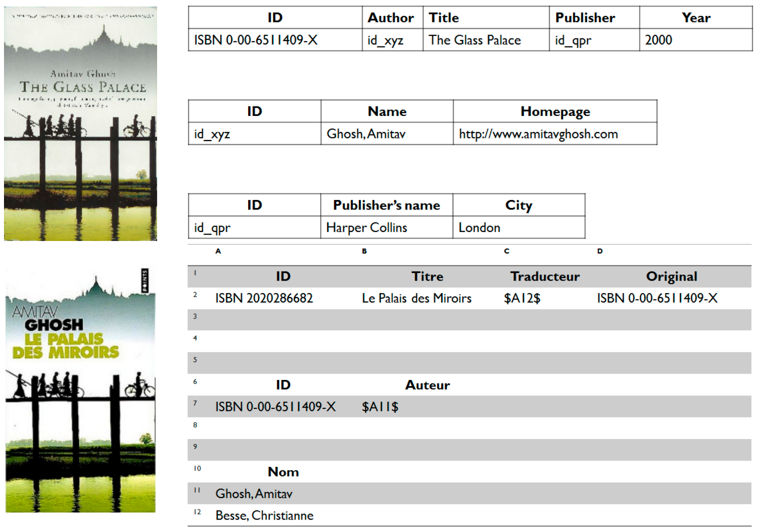
hence allowing her/him to concentrate on the application logic while discarding the details of a high-performance, complex computational framework.

In the context of data processing and query optimization issues for MapReduce, *Bloom filters* [35] and *Snappy data compression* [36] play a leading role as they are very often used to further improve the efficiency of MapReduce. Indeed, they are not only conceptually-related to MapReduce but also very relevant for our research, as they are finally integrated in our implemented framework (see Section 6). Bloom filters are *space-efficient probabilistic data structures* initially thought to test whether an element is member of a set or not. Bloom filters have a 100% *recall rate*, since false positive matches are possible, whereas false negative matches are not possible. It has been demonstrated that Bloom filters are very useful tools within MapReduce tasks, since, given the fact that they avoid false negatives, they allow for getting rid of irrelevant records during *map* phases of MapReduce tasks, hence configuring themselves as a very reliable space-efficient solution for MapReduce. Snappy data compression is a *fast C++ data compression and decompression library* proposed by *Google*. It does not aim at maximizing compression, but, rather, it aims at achieving a very high compression speed at a reasonable compression ratio. Given its orthogonality, Snappy has been used not only with MapReduce but also in other Google projects like BigTable.

4. RDF Graphs: Definitions, Concepts, Examples

Since RDF graphs play a central role in our research, in this Section, we focus in a greater detail on definitions, concepts and algorithms of such very important data structures of the popular Semantic Web. RDF is a Semantic Web data model oriented to represent *information on resources available on the Web*. To this end, a critical point is represented by the fact that RDF model *metadata* about such Web resources, which can be defined as data that describe other data. Thanks to metadata, RDF can easily implements some very important functionalities over the Semantic Web, such as: resource representation, querying resource on the Web, discovering resources on the Web, resource indexing, resource integration, interoperability among resources and applications, and so forth. RDF is primarily meant for managing resources on the Web for applications, not for end-users, as to support *querying the Semantic Web* (e.g., [37]), *Semantic Web interoperability* (e.g., [38]), *complex applications* (e.g., [39]), and so forth.

Looking into more details, RDF is founded on the main assertion that resources are identified on the Web via suitable *Web identifiers* like *Uniform Resource Identifiers* (URI) and they are described in terms of *property-value* pairs via *statements* that specify these properties and values. In an RDF statement, the *subject* identifies the resource (to be described), the *predicate* models the property of the subject, and the *object* reports the value of that property. Therefore, each RDF resource on the Web is modeled in terms of the triple (*subject*, *predicate*, *object*). This way, since Web resources are typically connected among them, the Semantic Web represented via RDF easily generates the so-called RDF graphs. In such graphs, RDF models statements describing Web resources via nodes and arcs. In particular, in this graph-ware model, a statement is represented by: (*i*) a node for the subject; (*ii*) a node for the object; (*iii*) an arc for the predicate, which is directed from the node modeling the subject to the node modeling the object. In terms of the popular *relational model*, an RDF statement can be represented like a tuple (*subject*, *predicate*, *object*) and, as a consequence, an RDF graph can be naturally represented via a *relational schema* storing information on subjects, predicated and objects of the corresponding Web resources described by the RDF statements modeled by that graph (e.g., [40]).


Amitav Ghosh
THE GLASS PALACE
AMITAV
GHOSH
LE PALAIS
DES MIROIRS
POINTS

ID	Author	Title	Publisher	Year
ISBN 0-00-6511409-X	id_xyz	The Glass Palace	id_qpr	2000

ID	Name	Homepage
id_xyz	Ghosh, Amitav	http://www.amitavghosh.com

ID	Publisher's name	City
id_qpr	Harper Collins	London

	A	B	C	D
1	ID	Titre	Traducteur	Original
2	ISBN 2020286682	Le Palais des Miroirs	$A12$	ISBN 0-00-6511409-X
3				
4				
5				
6	ID	Auteur		
7	ISBN 0-00-6511409-X	$A11$		
8				
9				
10	Nom			
11	Ghosh, Amitav			
12	Besse, Christianne			

Figure 2. Different Web media for the same book (**left**); and their corresponding relational schemas (**right**) [41].

Figure 2, borrowed from [41], shows an example where information on the Web regarding the book *"The Glass Palace"*, by A. Ghosh, occurs in two different sites in the vest of different media, one about the English version of the book, and the other one about the French version of the book. According to this, the two different Web resources are represented by two different relational schemas (see Figure 2). Figure 3, instead, shows the corresponding RDF graphs for the two different Web resources, respectively.

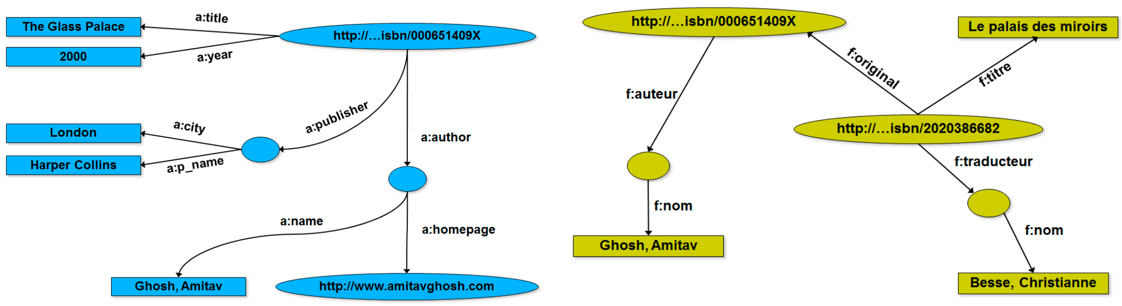


Figure 3. The two RDF graphs corresponding to the two different Web media for the same book reported in Figure 2 [41].

As highlighted before, one of the goals of RDF graphs is allowing the easy integration of correlated Web resources. This is, indeed, a typical case of Web resource integration because the two RDF graphs, even if different, describe two different media of the same resource. It is natural, as a consequence, to integrate the two RDF graphs in a unique (RDF) graph, even taking advantage from the "natural" topological nature of graphs, as shown in Figure 4. In particular, the integration process is driven by recognizing the same URI that identifies the book resource.

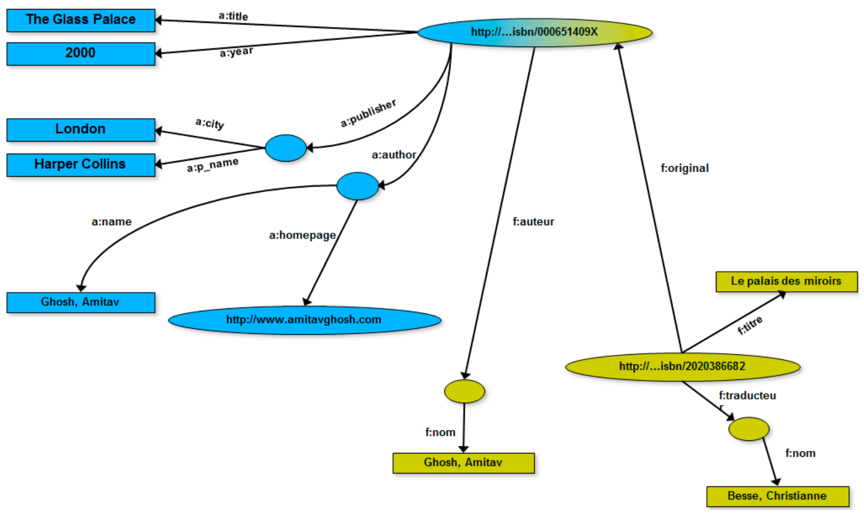


Figure 4. RDF graph obtained via integrating the two RDF graphs of Figure 3 [41].

Another nice property of the RDF graph data model that makes it particularly suitable to Semantic Web is its "open" nature, meaning that any RDF graph can be easily integrated with *external knowledge* on the Web about resources identified by the URIs it models. For instance, a common practice is that of integrating RDF graph with Web knowledge represented by *Wikipedia* [42] via ad-hoc tools (e.g., *DBpedia* [43]). This contributes to move the actual Web towards the Web of data and knowledge. Figure 5 shows the RDF graph of Figure 4 integrated with knowledge extracted from Wikipedia via DBpedia.

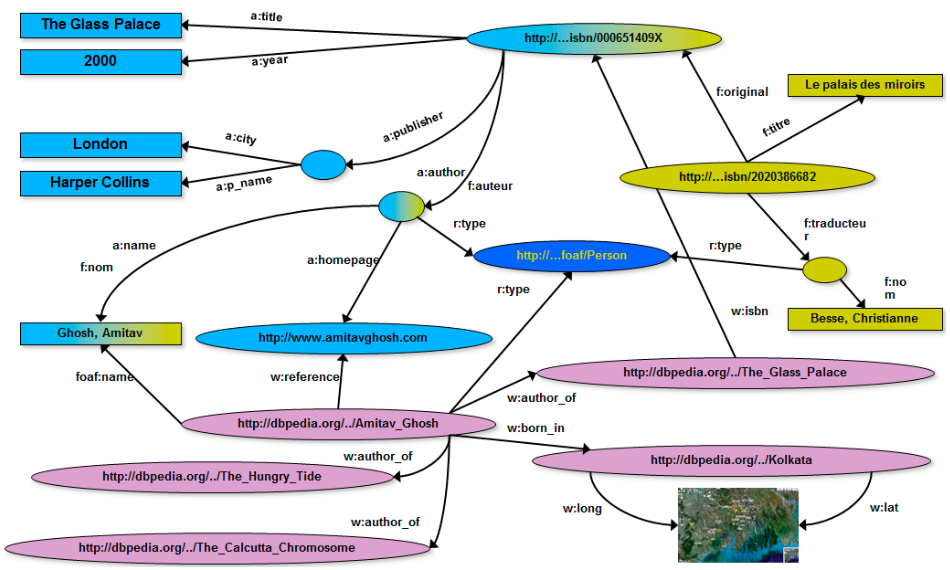


Figure 5. RDF graph obtained via integrating the RDF graph of Figure 4 with Wikipedia knowledge extracted via DBpedia [41].

It is worth noticing that, when RDF graphs are processed, applications can easily extract knowledge from them via so-called *RDF query language* [44] or other AI-inspired approaches (e.g., [45]). In addition to this, computing traversal paths over such graphs is critical for a wide range of Web applications. To give an example, traversal paths can be used as "basic" procedures of powerful *analytics over Web Big Data* (e.g., [46]). This further gives solid motivation to our research.

5. Related Work

The conceptual and literature context of our research includes the following areas:

- Parallel BFS-based and DFS-based traversal strategies;
- MapReduce algorithms for big data processing;
- MapReduce algorithms over RDF databases;
- MapReduce algorithms over RDF graphs.

In the following, we provide a discussion of main approaches falling in the four distinct areas.

5.1. Parallel BFS-Based and DFS-Based Traversal Strategies

The basic problem we investigate in our research is strictly related to the issue of supporting parallel BFS-based and DFS-based traversal computing, as to gain into efficiency. Here, several proposals appear in literature, also based on the *Depth-First Search* (DFS) strategy, which is an alternative to BFS. In the following, we mention some noticeable ones. As regards BFS-based alternatives, [47] inspects the problem of supporting BFS-based traversals of *trees* in parallel. The proposed algorithm is *cost-optimal*, and it is based on the innovative vision of interpreting the BFS-based traversal problem into a *parentheses matching problem*. Performance analysis is promising as well. [48] instead introduces a *parallel external-memory algorithm* for supporting BFS-based traversal computing based on a *cluster of workstations*. The proposed algorithm has the merit of distributing the input workload according to intervals that are computed at runtime via a *sampling-based process*. An interesting experimental analysis focusing on how the operating system supports the caching of external memory into internal memory while running the algorithm is also proposed. Finally, as regards DFS-based alternatives, [49] proposes a modified version of [47], still focusing on trees, where the main algorithm is adapted as to work with the DFS's principle.

5.2. MapReduce Algorithms for Big Data Processing

Paper [50] puts light on main issues and challenges of big data processing over MapReduce, by highlighting actual data management solutions that found over this computational platform. Several aspects are touched, including *job optimization*, *physical data organization*, *data layouts*, *indexes*, and so forth. Finally, a comparative analysis between Hadoop-MapReduce and *Parallel DBMS* is provided, by highlighting their similarities and differences.

Contribution [51] *investigates interactive analytical processing in big data systems*, with particular emphasis over the case of MapReduce, even considering the industrial applicative setting. In more detail, authors capture and model typical *big data processing workloads*, which are characterized by several small, short and increasingly interactive jobs, which clearly contrast with the large, long-running batch jobs for which MapReduce was originally designed. In line with this trend, authors provide an empirical analysis of MapReduce traces from several business-critical deployments inside Facebook and Cloudera instances in a wide range of applications scenarios ranging from e-commerce to telecommunication systems, from social media to large-scale retail systems, and so forth. As a result of their empirical analysis, authors provide a new class of MapReduce workloads that are composed by *interactive analysis components* making heavy use of query-like programming frameworks on top of MapReduce.

Paper [52] introduces and experimentally assesses H_2RDF, an innovative, *fully-distributed RDF framework* that *combines MapReduce with NoSQL (distributed) data stores*. With respect to the state-of-the-art contributions, H_2RDF allows two distinctive characteristics to be achieved: (*i*) high efficiency in both *simple* and *multi-join SPARQL queries* [53] via joins that execute on the basis of query selectivity; (*ii*) alternatively-available centralized and MapReduce-based join execution that ensures higher speed-up during query evaluation. The system is targeted to billions of RDF triples even with small commodity Clouds, and its performance over that of comparative methods is experimentally proven.

Contribution [54] similarly focuses the attention on the problem of *efficiently supporting SPARQL join queries over big data in Cloud environments via MapReduce*. The novelty of this research effort relies on an implementation of the *distributed sort-merge join algorithm* on top of MapReduce where the join is computed during the map phase completely. The problem of *cascaded executions* that may derive from executing multiple joins is addressed during the reduce phase by ensuring that the "right-hand" side of the join is always pre-sorted on the required attributes. Experiments show an excellent performance over comparative approaches.

Finally, at the mere system-side, several works focus the attention on very interesting applicative settings, such as the case of supporting MapReduce-based big data processing on *multi-GPU systems* (e.g., [55]), and the case of efficiently supporting *GIS polygon overlay computation* with MapReduce for *spatial big data processing* (e.g., [56]).

5.3. MapReduce Algorithms for RDF Databases

The relevance of RDF query processing via MapReduce has been highlighted by several studies (e.g., [57–59]). Basically, these works focus the attention on RDF data management architectures and systems designed for Cloud environments, with a special focus to large-scale RDF processing. Approaches investigated here mainly consider several special tasks that can be performed on the Web via RDF, such as *Web data publishing* and *Web data exchanging*.

Paper [60] focuses the attention on *querying linked data based on MapReduce*. Linked data can easily be modeled via RDF databases. In particular, authors consider linked data that are arbitrarily partitioned and distributed over a set of Cloud nodes, such that input queries are decomposed into a set of suitable *sub-queries*, each one involving data stored in a specific node. The proposed method is two-step in nature. In the first step, sub-queries are executed on the respective node in an isolated manner. In the second step, intermediate results are combined in order to obtain the final answer to the input query. The innovation introduced by this research effort consists in the fact that the proposed query algorithm is independent of all the parameters of the model, *i.e.*, linked data partitioning, local data storage, query decomposition mechanism, local query algorithm.

SPARQL query processing over RDF data has attracted a lot of attention from the research community as well. In this context, several approaches have been proposed recently. [61] addresses the problem of *supporting SPARQL queries over very large RDF datasets* via mapping SPARQL statements over *PigLatin* [62] programs. These programs are finally executed in terms of a series of MapReduce jobs on top of a Hadoop cluster. The study also provides an interesting experimental trade-off analysis on top of a popular benchmark dataset. [63] also investigates *scalability issues of processing SPARQL queries over Web-scale RDF knowledge bases*. To this end, authors propose an innovative partitioning method particularly suitable to RDF data that introduces the nice amenity of providing *effective indexing schemes* for supporting efficient SPARQL query processing over MapReduce. Still in this area, [64,65] provide anatomy and main functionalities of *a semantic data analytical engine* and *a Cloud-enabled RDF triple store*, respectively. Both embed several points of research innovation over state-of-the-art proposals.

Contribution [66] studies how to apply *compression paradigms* to obtain *scalable RDF processing with MapReduce*. Authors point the specific case of Semantic Web, where *many billions of statements*, which are released using the RDF data model, exist. Therefore, they propose to apply a novel data compression technique in order to tame the severity of data size, and thus design *a set of distributed MapReduce algorithms* to efficiently compress and decompress a large amount of RDF data. In particular, they apply a *dictionary encoding technique* that maintains the structure of the (Web) data. The solution is implemented as a prototype using the Hadoop framework, and its performance is evaluated over a large amount of data and scales linearly on both input size and number of nodes.

Still in the context of optimization issues, [67] proposes *a novel nested data model for representing intermediate data* of RDF processing concisely using *nesting-aware dataflow operators* that allow for lazy and partial un-nesting strategies. Indeed, authors correctly recognize that, during RDF data processing, intermediate results of joins with multi-valued attributes or relationships contain redundant sub-tuples due to repetition of single-valued attributes. As a consequence, the amount of redundant content is high for real-world multi-valued relationships in typical data-intensive instances such as social networks or biological datasets. Unfortunately, in MapReduce-based platforms, redundancy in intermediate results contributes avoidable costs to the overall I/O, sorting, and network transfer overhead of join-intensive workloads due to longer workflows. Therefore, in order to deal with this challenge, the proposed approach reduces the overall I/O and network footprint of a workflow by concisely representing intermediate results during most of a workflow's execution, until complete un-nesting is absolutely necessary.

Finally, [68] focuses the attention on query optimization aspects of massive *data warehouses* over RDF data. In this context, a relevant problem is represented by the evaluation of join queries over RDF data *when triple patterns with unbound properties occur*. A clear example of this case is represented by edges with “don’t” care labels. As a consequence, when evaluating such queries using relational joins, intermediate results contain *high redundancy* that is empathized by such unbound properties. In order to face-off this problem, authors propose *an algebraic optimization technique that interprets unbound-property queries on MapReduce via using a non-relational algebra based on a TripleGroup data model*. Novel logical and physical operators and query rewriting rules for interpreting unbound-property queries using the TripleGroup-based data model and algebra are formally introduced. The proposed framework is implemented on top of *Apache Pig* [69], and experiments on both synthetic and real-word benchmark datasets that show the benefits of the framework are provided and discussed in details.

5.4. MapReduce Algorithms for RDF Graphs

Paper [70] focuses the attention on the problem of effectively and efficiently supporting scalable storage and retrieval of large volumes of in-memory-representations of RDF graphs by exploiting a combination of MapReduce and HBase. The solution conveys the so-called RDFChain framework, whose storage schema reflects all the possible join query patterns, thus providing a reduced number of storage accesses on the basis of the joins embedded in the target queries. In addition to this, a cost-based map-side join of RDFChain is provided. This allows for reducing the number of map jobs significantly.

Contribution [71] considers the research challenge of *optimizing RDF graph pattern matching on MapReduce*. Authors correctly recognize that the MapReduce computation model provides limited *static optimization techniques* used in relational DBMS, such as indexing and cost-based optimizations, hence it is mandatory to investigate *dynamic optimization techniques* for join-intensive tasks on MapReduce. With this idea in mind, authors extend their previous *Nested Triple Group data model and Algebra* (NTGA) for efficient graph pattern query processing in the Cloud [72] and achieve the definition of a *scan-sharing technique* that is used to optimize the processing of graph patterns with repeated properties. Specifically, the proposed scan-sharing technique eliminates the need for repeated scanning of input relations when properties are used repeatedly in graph patterns.

Finally, [73] proposes *a cost-model-based RDF join processing solution using MapReduce* to minimize the query responding time as much as possible. In particular, authors focus the attention on *SPARQL queries in a shared-nothing environment on top of MapReduce*, and propose a technique based on which they (1) first transform a SPARQL query into a sequence of MapReduce jobs; and (2) then create a novel index structure, called *All Possible Join tree* (*APJ-tree*) for reducing the searching space inducted by the optimal execution plan of a set of MapReduce jobs. To speed up the join processing, they employ *hybrid join* and Bloom filters for performance optimization. Authors complete their analytical contributions by means of an extensive set of experiments on real data sets that prove the effectiveness of the proposed cost model.

6. BFS-Based Implementation of RDF Graph Traversals over MapReduce

There exist in literature two well-known approaches for representing a graph: through an *adjacency matrix*, and through *adjacency lists*. The former has the advantage of $O(1)$ constant lookup over graph data, with the drawback of an increased usage of memory ($n \times n$, where n represents the number of vertices), also for sparse graphs. In practice, while the majority of graphs are not so dense, we end with a waste of space having a big matrix with few values (not all the cells of the matrix will be used to store a value). The latter stores, for every vertex, a list of vertices connected to it (called *adjacent nodes*), thus reducing the overall memory requirement. This representation has also the advantage of being suited to feed a MapReduce job. As regards the proper RDF graph implementation as *Abstract Data Type* (ADT), we exploited the well-known *Apache Jena RDF API* [74], which models the target RDF graph in terms of collection of *RDF nodes*, which are connected via a *direct graph* by means of *labelled relations*.

There are several algorithms for visiting a graph, but BFS (see Algorithm 1) has been proven to be more suited for parallel processing. BFS algorithm cannot be expressed into a single MapReduce job directly, because the `while` construct at every step increments the *frontier* of exploration in the graph. Moreover, this frontier expansion permits the finding of the paths to *every* node explored from the source node. Since the BFS algorithm cannot be codified within a MapReduce job, we propose a novel approach to coordinate a series of MapReduce jobs that check when to stop the execution of these jobs. This is called *convergence criterion*. To this end, we propose feeding the output of a job as the input of the successive one, and to apply a way to count if further nodes must be explored via maintaining a *global counter* updated at each iteration that, in the standard BFS, is represented by the iterator's position in the queue itself. In our proposed implementation, the only convergence criterion that we make use of is thus represented by counting the jobs that terminate (to this end, MapReduce's *DistributedCache* is exploited), but of course other more complex criteria may be adopted in future research efforts (e.g., based on *load balancing issues*).

Algorithm 1: BFS(u, n, L)

```
Input: $u$ -- source node from which the visit starts; $n$ -- number of vertices;
$L$ -- adjacency lists
Output: $Q$ -- list of visited nodes

Begin
$Q \leftarrow \emptyset$;
$visited \leftarrow \emptyset$;
for ($i$ = 0 to $n$ -- 1) do
$visited[i]$ = 0;
endfor;
$visited[u]$ = 1;
$Q \leftarrow u$;
while ($Q \neq \emptyset$) do
$u \leftarrow Q$;
for (each $v \in L_u$) do     // $L_u$ is the adjacency list of $u$
  if ($visited[v]$ == 0) then
    $visited[v]$ = 1;
    $Q \leftarrow v$;
  endif;
endfor;
endwhile;
return $Q$;
End;
```

Basically, in order to control the algorithmic flow and safely achieve the convergence criterion, the proposed algorithm adopts the well-known *colorability programming metaphor*. According to this programming paradigm, graph processing algorithms can be implemented by assigning a color-code to each possible node's state (in the *proper* semantics of the target algorithm) and then superimposing certain actions for each specific node's state. This paradigm is, essentially, an enriched *visiting strategy* over graphs, and, in fact, BFS is only a possible instance of such strategy. To this end, the proposed approach makes use of suitable *metadata information* (stored into text files), which models, for each node of the graph stored in a certain Cloud node, its adjacent list plus additional information about the node's state. An example of these metadata is as follows:

- `1 3,2, | GREY | 1`
- `10 12, | WHITE | ?`
- `11 10, | WHITE | ?`
- `12 NULL, | WHITE | ?`
- `13 NULL, | WHITE | ?`
- `14 6,12, | WHITE | ?`

In more detail, for each row, stored metadata have the following semantics: (see the previous example—the description is field-based): (*i*) the first number is the node ID; (*ii*) the subsequent list of numbers represents the adjacent list for that node, and it is equal to NULL if empty; (*iii*) the subsequent color-code represents the state of the node; (*iv*) the last symbol states if the target node has been processed (*i.e.*, then, it is equal to "1") or not (*i.e.*, then, it is equal to "?"). The adoption of this metadata information exposes the clear advantage of supporting *portability* and *interoperability* within Cloud environments widely.

Looking into code-wise details, the proposed algorithm consists of a series of iterations where each iteration represents a MapReduce job using the output from the previous MapReduce job until the convergence criterion is met. The *Mapper* job (see Algorithm 2) allows us to expand the frontier of exploration by selecting the new nodes to explore (the ones that in the standard BFS algorithm are inserted into the queue) and the ones already explored (or still not explored), as to complete the output.

Algorithm 2: Map(*key*, *value*)

```
Input: key -- ID of the current node; value -- node information
Output: void -- a new node is emitted

Begin
u ← (Node)value;
if (u.color == GREY) then
for (each v ∈ $L_u$) do     // $L_u$ is the adjacency list of u
  v ← new Node();
  v.color == GREY;     // mark v as unexplored
  for (each c ∈ $P_u$) do     // $P_u$ is the path list of u
    c.ID = u.ID;
    c.tag = u.tag;
    v.add(c);
  endfor;
  emit(v.ID,v);
endfor;
u.color = BLACK;     // mark u as explored
endif;
emit(key,u);
return;
End;
```

The *Reducer* job (see Algorithm 3) allows us to (*i*) merge the data provided by the *Mapper* and re-create an adjacency list that will be used at the next step and (*ii*) update the counter for the convergence test. More in detail, in order to merge nodes with the same ID, a color priority method was devised so that, in the case of the generation of a node with the same ID with different color, the darkest one will be saved.

Algorithm 3: Reduce(*key*, *values*)

```
Input: key -- ID of the current node; values -- node information list
Output: void -- updates for the current node are emitted

Begin
newNode ← new Node();
for (each u ∈ values) do
    merge(newNode.edgeList(),u.edgeList());
      // merge the newNode's edge list with u's edge list
    merge(newNode.tagList(),u.tagList());
     // merge the newNode's tag list with u's tag list
    newNode.setToDarkest(newNode.color,u.color);
     // set color of newNode to the darkest color
endfor;                                       // between the newNode's color and the u's color
update(WHITE,GREY,BLACK);
// update the number of processed nodes having color WHITE, GREY, BLACK
emit(key.newNode);     // emit updates for the current node
return;
End;
```

By inspecting our proposed algorithm, it clearly turns that new nodes are generated via a series of MapReduce steps. This may increase *communication costs*, but our experimental analysis proves that they are limited in impact. Indeed, in Section 7, we show that our throughput performance analysis, which includes the impact of communication costs, exposes a good behavior. This demonstrates that communication costs do not impact on the overall performance in a relevant percentage amount. Despite this, further optimizations of communication routines should be considered as future work.

Finally, it should be noted that, while our proposed approach is general enough for being applied to *any* kind of graph-like data (e.g., social networks, sensor networks, and so forth), it is specially focused on RDF graphs due to the prevalence of such graph-like data on the Web, as a major form of big Web data. Indeed, our main aim is that of taking advantage from the efficiency of MapReduce as to cope with the *volume* of big Web data, following the strict requirements dictated by recent applications dealing with this kind of data (e.g., [75,76]).

7. Experimental Analysis and Results

In this Section, we illustrate our experimental assessment and analysis that we devised in order to demonstrate the feasibility of our approach, and the effectiveness and the efficiency of our algorithms. The experimental environment used for our experiments is depicted in Figure 6. In particular, the target cluster comprises five hosts having the hardware described in Table 1.

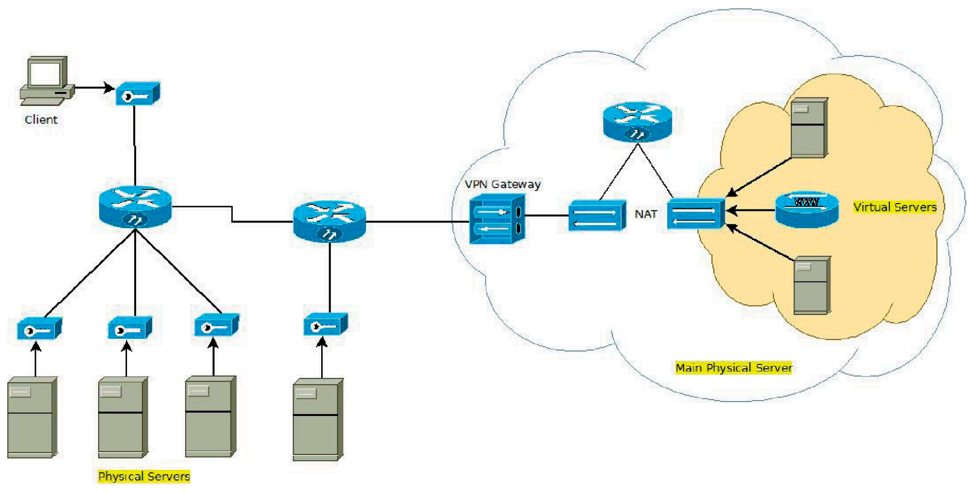


Figure 6. System environment.

Table 1. Hardware in the cluster of the experimental environment.

Core Num.	Core Speed (GHz)	HDD	RAM (GB)
6	2.4	1	6
8	2.4	1	2
4	2.7	1	4
4	2.7	1	4
4	2.7	1	4

In our experimental campaign, we ran tests on top of our cluster in order to verify its functionality and throughput first. These tests give us a baseline that was helpful to tune the cluster through the various configuration options and the schema of tables themselves. *The Yahoo! Cloud Serving Benchmark* (YCSB) [77] was used to run comparable workloads against different storage systems. While primarily built to compare various systems, YCSB is also a reasonable tool for performing an HBase cluster burn-in or performance-oriented tests (e.g., [78]). To this end, it is necessary to create some tables and load a bunch of test data (selected by the user or randomly generated) into these tables (generally, at least 10^8 rows of data should be loaded). In order to load data, we used the YCSB client tool [77] by generating a synthetic dataset that comprises 10^{12} rows. After loading the data, YCSB permits to run tests with the usage of the CRUD operations in a set of "workloads". In the YCSB's semantics, workloads are configuration files containing the percentage of operations to do, how many operations, how many threads should be used, and other useful experimental parameters.

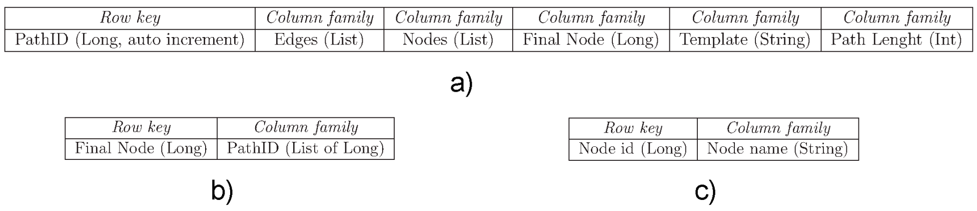
We ran tests on both the default configuration and other tentative configurations by reaching a throughput of 1300 Op/s for reading-like operations and 3500 Op/s for writing-like operations, thus increasing by 50% the performance obtained with the default configurations. It is worth mentioning that, using both the Snappy compression scheme for reducing the size of data tables and Bloom filters for implementing the membership test in the adjacent lists (which may become very large in real-life RDF graphs) contributed in the performance gain heavily. Table 2 reports the experimental results provided by this experiment, which comprises the impact of communication costs.

Table 2. Throughput performance.

	Reading-Like Operations	**Writing-Like Operations**
Throughput (Op/s)	1300	3500

After the first series of experiments devoted to assess the throughput of the framework, we focused our attention on the data management capabilities of the framework by introducing an efficient schema and an efficient client capable of uploading and retrieving data efficiently, *i.e.*, based on the effective schema designed. The first solution contemplated an RDBMS(*Relational DataBase Management System*)-like approach to the schema design while the second one was more akin to NoSQL paradigms. As it will be mentioned later, the second approach is the one that exposes the best during the retrieval phase, lagging behind the first approach for data uploading. Due to the fact that the performance difference, however, was negligible, the second approach was the one to be implemented. To both solutions, all the optimizations about performance increasing discovered during the "load test" phase were applied (*i.e.*, table data Snappy-based compression and Bloom filters).

Therefore, we finally implemented three tables for HBase data storage (see Figure 7). As shown in Figure 7, the following tables have been devised: (i) table Path, shown in Figure 7a, is a repository of paths containing all the data modeling a path (*i.e.*, *Row key*, *Column family*, *etc.*); (*ii*) for each sink, table Sink-Path, shown in Figure 7b, contains the list of the path IDs (related to the first table) that ends with that node; (*iii*) for each node, table Node-Label, shown in Figure 7c, contains the labels of each node.



Row key	*Column family*	*Column family*	*Column family*	*Column family*	*Column family*
PathID (Long, auto increment)	Edges (List)	Nodes (List)	Final Node (Long)	Template (String)	Path Lenght (Int)

a)

Row key	*Column family*
Final Node (Long)	PathID (List of Long)

b)

Row key	*Column family*
Node id (Long)	Node name (String)

c)

Figure 7. HBase configuration for the running experiment: (**a**) table Path; (**b**) table Sink-PathID; (**c**) table Node-Label.

On top of this HBase data storage, we ran simple operations like Get, Batch and Scan, of course being the framework running our proposed algorithm. This allowed us to retrieve one million (1M) of rows within 5.5 ms through *cold-cache* experiments and within 1.3 ms through *warm-cache* experiments. Table 3 reports these results.

Table 3. Data management performance.

	Cold-Cache Mode	Warm-Cache Mode
Retrieval Time for 1 M Rows (ms)	5.5	1.3

Finally, by analyzing our experimental results, we conclude that our framework for computing BSF-based traversals of RDF graphs clearly exposes both effectiveness and efficiency. As future extension, in order to further improve performance, the proposed framework could be implemented on top of the well-known *Apache Spark analytical engine* [79], as Spark exposes an optimized in-memory computation scheme for large-scale data processing. This would allow us to improve response times.

8. Conclusions and Future Work

In this paper, we propose an effective and efficient framework for computing BFS-based traversals of large-scale RDF graphs over MapReduce-aware platforms. Our approach embeds intelligent algorithms, and it turns out to be extremely useful in the emerging context of so-called big Web data. We have also provided experiments that clearly demonstrate the speed-up ensured by our solutions over comparative approaches. This offers exciting possibilities towards the support of *RDF graph analysis and mining methodologies* (e.g., [80,81]) over large-scale data sets, thanks to the powerful run-time support offered by MapReduce.

Future work is oriented towards the integration of our algorithms with Cloud-inspired frameworks devoted to the distributed management of large-scale data repositories via data compression metaphors (e.g., [82,83]).

Author Contributions: Joint work – all the authors work together and contribut equally.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dean, J.; Ghemawat, S. MapReduce: Simplified Data processing on Large Clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
2. Ebay Data Warehouses. Available online: http://www.dbms2.com/2009/04/30/ebays-two-enormous-data-warehouses/ (accessed on October 31 2015).
3. Facebook Hadoop and Hive. Available online: http://www.dbms2.com/2009/05/11/facebook-hadoop-and-hive/ (accessed on October 31 2015).
4. Facebook. Available online: http://developers.facebook.com/ (accessed on October 31 2015).
5. MySpace. Available online: http://wiki.developer.myspace.com/index.php?title=Main_Page (accessed on October 31 2015).
6. NetFlix Documentation. Available online: http://developer.netflix.com/docs (accessed on October 31 2015).
7. Leskovec, J.; Kleinberg, J.M.; Faloutsos, C. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, Chicago, IL, USA, 21–24 August 2005; pp. 177–187.
8. Bahmani, B.; Kumar, R.; Vassilvitskii, S. Densest Subgraph in Streaming and MapReduce. *Proc. VLDB Endow.* **2012**, *5*, 454–465. [CrossRef]
9. Zhong, N.; Yau, S.S.; Ma, J.; Shimojo, S.; Just, M.; Hu, B.; Wang, G.; Oiwa, K.; Anzai, Y. Brain Informatics-Based Big Data and the Wisdom Web of Things. *IEEE Intell. Syst.* **2015**, *30*, 2–7. [CrossRef]
10. Lane, J.; Kim, H.J. Big Data: Web-Crawling and Analysing Financial News Using RapidMiner. *Int. J. Bus. Inf. Syst.* **2015**, *19*, 41–57. [CrossRef]
11. W3C. RDF 1.1 Concepts and Abstract Syntax—W3C Recommendation 25 February 2014. Available online: http://www.w3.org/TR/rdf11-concepts/ (accessed on October 31 2015).
12. Cappellari, P.; Virgilio, R.D.; Roantree, M. Path-Oriented Keyword Search over Graph-Modeled Web Data. *World Wide Web* **2012**, *15*, 631–661. [CrossRef]

13. Bröcheler, M.; Pugliese, A.; Subrahmanian, V.S. Dogma: A Disk-Oriented Graph Matching Algorithm for RDF Databases. In *The Semantic Web—ISWC*; Springer: Berlin Heidelberg, Germany, 2009; pp. 97–113.
14. Fan, W.; Li, J.; Ma, S.; Tang, N.; Wu, Y.; Wu, Y. Graph Pattern Matching: From Intractable to Polynomial Time. *Proc. VLDB Endow.* **2010**, *3*, 264–275. [CrossRef]
15. Zhang, S.; Yang, J.; Jin, W. Sapper: Subgraph Indexing and Approximate Matching in Large Graphs. *Proc. VLDB Endow.* **2010**, *3*, 1185–1194. [CrossRef]
16. Yu, B.; Cuzzocrea, A.; Jeong, D.H.; Maydebura, S. On Managing Very Large Sensor-Network Data Using Bigtable. In Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Ottawa, ON, Canada, 13–16 May 2012; pp. 918–922.
17. Yu, B.; Cuzzocrea, A.; Jeong, D.; Maybedura, S. A Bigtable/MapReduce-based Cloud Infrastructure for Effectively and Efficiently Managing Large-Scale Sensor Networks. In *Data Management in Cloud, Grid and P2P Systems*; Springer: Berlin Heidelberg, Germany, 2012; pp. 25–36.
18. Hadoop. Available online: http://wiki.apache.org/hadoop (accessed on October 31 2015).
19. Cuzzocrea, A.; Furfaro, F.; Mazzeo, G.M.; Saccà, D. A Grid Framework for Approximate Aggregate Query Answering on Summarized Sensor Network Readings. In Proceedings of the OTM Confederated International Workshops and Posters, GADA, JTRES, MIOS, WORM, WOSE, PhDS, and INTEROP 2004, Agia Napa, Cyprus, 25–29 October 2004; pp. 144–153.
20. Cuzzocrea, A.; Furfaro, F.; Greco, S.; Masciari, E.; Mazzeo, G.M.; Saccà, D. A Distributed System for Answering Range Queries on Sensor Network Data. In Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops, 2005. PerCom 2005 Workshops, Kauai Island, HI, USA, 8–12 March 2005; pp. 369–373.
21. Cuzzocrea, A. Data Transformation Services over Grids With Real-Time Bound Constraints. In *On the Move to Meaningful Internet Systems: OTM 2008*; Springer: Berlin Heidelberg, Germany, 2008; pp. 852–869.
22. Ghemawat, S.; Gobioff, H.; Leung, S.T. The Google Fle System. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA, 19–22 October 2003; pp. 29–43.
23. Apache Nutch. Available online: http://nutch.apache.org/ (accessed on October 31 2015).
24. Amazon. Available online: http://www.amazon.com (accessed on October 31 2015).
25. Elastic MapReduce Web Service. Available online: http://aws.amazon.com/elasticmapreduce/ (accessed on October 31 2015).
26. Amazon Elastic Compute Cloud—EC2. Available online: http://wiki.apache.org/hadoop/AmazonEC2 (accessed on October 31 2015).
27. NetFlix. Available online: https://www.netflix.com/ (accessed on October 31 2015).
28. Hulu. Available online: http://www.hulu.com/ (accessed on October 31 2015).
29. HBase—Apache Software Foundation Project Home Page. Available online: http://hadoop.apache.org/hbase/ (accessed on October 31 2015).
30. Abadi, D.J.; Boncz, P.A.; Harizopoulos, S. Column oriented Database Systems. *Proc. VLDB Endow.* **2009**, *2*, 1664–1665. [CrossRef]
31. Cattell, R. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.* **2010**, *39*, 12–27. [CrossRef]
32. Shvachko, K.; Kuang, H.; Radia, S.; Chansler, R. The Hadoop Distributed File System. In Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, USA, 3–7 May 2010; pp. 1–10.
33. Chang, F.; Dean, J.; Ghemawat, S.; Hsieh, W.C.; Wallach, D.A.; Burrows, M.; Chandra, T.; Fikes, A.; Gruber, R.E. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* **2008**, *26*. [CrossRef]
34. Lin, J.; Dyer, C. Data-Intensive Text Processing with MapReduce. In *Synthesis Lectures on Human Language Technologies*; Morgan & Claypool Publishers: San Rafael, CA, USA, 2010.
35. Bloom, B.H. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* **1970**, *13*, 422–426. [CrossRef]
36. Snappy: A Fast Compressor/Decompressor. Available online: https://google.github.io/snappy/ (accessed on October 31 2015).
37. Broekstra, J.; Kampman, A.; van Harmelen, F. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *The Semantic Web—ISWC*; Springer: Berlin Heidelberg, Germany, 2002; pp. 54–68.

38. Decker, S.; Melnik, S.; van Harmelen, F.; Fensel, D.; Klein, M.C.A.; Broekstra, J.; Erdmann, M.; Horrocks, I. The Semantic Web: The Roles of XML and RDF. *IEEE Intern. Comput.* **2000**, *4*, 63–74. [CrossRef]
39. Beckett, D.J. The Design and Implementation of the Redland RDF Application Framework. *Comput. Netw.* **2002**, *39*, 577–588. [CrossRef]
40. Huang, J.; Abadi, D.J.; Ren, K. Scalable SPARQL Querying of Large RDF Graphs. *Proc. VLDB Endow.* **2011**, *4*, 1123–1134.
41. Herman, I. Introduction to Semantic Web Technologies. *SemTech*, 2010. Available online: http://www.w3.org/2010/Talks/0622-SemTech-IH/ (accessed on October 31 2015). – material redistributed under the Creative Common License (http://creativecommons.org/licenses/by-nd/3.0/ – accessed on October 31, 2015).
42. Wikipedia. Available online: https://www.wikipedia.org/ (accessed on October 31 2015).
43. DBpedia. Available online: http://wiki.dbpedia.org/ (accessed on October 31 2015).
44. W3C. RDQL—A Query Language for RDF—W3C Member Submission 9 January 2004. Available online: http://www.w3.org/Submission/RDQL/ (accessed on October 31 2015).
45. Gabrilovich, E.; Markovitch, S. Computing Semantic Relatedness Using Wikipedia-based Explicit Semantic Analysis. In Proceedings of the 20th International Joint Conference on Artifical Intelligence, Hyderabad, India, 6–12 January 2007; pp. 1606–1611.
46. Chandramouli, N.; Goldstein, J.; Duan, S. Temporal Analytics on Big Data for Web Advertising. In Proceedings of the 2012 IEEE 28th International Conference on Data Engineering (ICDE), Washington, DC, USA, 1–5 April 2012; pp. 90–101.
47. Chen, C.C.Y.; Das, S.K. Breadth-First Traversal of Trees and Integer Sorting in Parallel. *Inf. Process. Lett.* **1992**, *41*, 39–49. [CrossRef]
48. Niewiadomski, R.; Amaral, J.N.; Holte, R.C. A Parallel External-Memory Frontier Breadth-First Traversal Algorithm for Clusters of Workstations. In Proceedings of the International Conference on Parallel Processing, Columbus, OH, USA, 14–18 August 2006; pp. 531–538.
49. Chen, C.C.Y.; Das, S.K.; Akl, S.G. A Unified Approach to Parallel Depth-First Traversals of General Trees. *Inf. Process. Lett.* **1991**, *38*, 49–55. [CrossRef]
50. Dittrich, J.; Quiané-Ruiz, J.A. Efficient Big Data Processing in Hadoop MapReduce. *Proc. VLDB Endow.* **2012**, *5*, 2014–2015. [CrossRef]
51. Chen, Y.; Alspaugh, S.; Katz, R.H. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *Proc. VLDB Endow.* **2012**, *5*, 1802–1813. [CrossRef]
52. Papailiou, N.; Tsoumakos, D.; Konstantinou, I.; Karras, P.; Koziris, N. H2RDF: Adaptive Query Processing on RDF Data in the Cloud. In Proceedings of the 21st International Conference on World Wide Web, Lyon, France, 16–20 April 2012; pp. 397–400.
53. W3C. SPARQL 1.1 Overview—W3C Recommendation 21 March 2013. Available online: http://www.w3.org/TR/sparql11-overview/ (accessed on October 31 2015).
54. Przyjaciel-Zablocki, M.; Schätzle, A.; Skaley, E.; Hornung, T.; Lausen, G. Map-Side Merge Joins for Scalable SPARQL BGP Processing. In Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom), Bristol, UK, 2–5 December 2013; pp. 631–638.
55. Jiang, H.; Chen, Y.; Qiao, Z.; Weng, T.H.; Li, K.C. Scaling Up MapReduce-based Big Data Processing on Multi-GPU Systems. *Clust. Comput.* **2015**, *18*, 369–383. [CrossRef]
56. Wang, Y.; Liu, Z.; Liao, H.; Li, C. Improving the Performance of GIS Polygon Overlay Computation with MapReduce for Spatial Big Data Processing. *Clust. Comput.* **2015**, *18*, 507–516. [CrossRef]
57. Kaoudi, Z.; Manolescu, I. RDF in the Clouds: A Survey. *VLDB J.* **2015**, *24*, 67–91. [CrossRef]
58. Rohloff, K.; Schantz, R.E. High-Performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The SHARD Triple-Store. In Proceedings of the Programming Support Innovations for Emerging Distributed Applications, Reno, NV, USA, 17 October 2010.
59. Ladwig, G.; Harth, A. CumulusRDF: Linked Data Management on Nested Key-Value Stores. In Proceedings of the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2011) at the 10th International Semantic Web Conference, Bonn, Germany, 23–27 October 2011; pp. 30–42.
60. Gergatsoulis, M.; Nomikos, C.; Kalogeros, E.; Damigos, M. An Algorithm for Querying Linked Data Using Map-Reduce. In Proceedings of the 6th International Conference, Globe 2013, Prague, Czech, 28–29 August 2013; pp. 51–62.

61. Schätzle, A.; Przyjaciel-Zablocki, M.; Lausen, G. PigSPARQL: Mapping SPARQL to Pig Latin. In Proceedings of the International Workshop on Semantic Web Information Management, Athens, Greece, 12–16 June 2011.
62. Olston, C.; Reed, B.; Srivastava, U.; Kumar, R.; Tomkins, A. Pig Latin: A Not-So-Foreign Language for Data Processing. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, 2 March 2008; pp. 1099–1110.
63. Nie, Z.; Du, F.; Chen, Y.; Du, C.; Xu, L. Efficient SPARQL Query Processing in MapReduce through Data Partitioning and Indexing. In *Web Technologies and Applications*; Springer: Berlin Heidelberg, Germany, 2012; pp. 628–635.
64. Du, J.H.; Wang, H.; Ni, Y.; Yu, Y. HadoopRDF: A Scalable Semantic Data Analytical Engine. In *Intelligent Computing Theories and Applications*; Springer: Berlin Heidelberg, Germany, 2012; Volume 2, pp. 633–641.
65. Punnoose, R.; Crainiceanu, A.; Rapp, D. Rya: A Scalable RDF Triple Store for the Clouds. In Proceedings of the 1st International Workshop on Cloud Intelligence, Istanbul, Turkey, 31 August 2012.
66. Urbani, J.; Maassen, J.; Drost, N.; Seinstra, F.J.; Bal, H.E. Scalable RDF Data Compression with MapReduce. *Concurr. Comput. Pract. Exp.* **2013**, *25*, 24–39. [CrossRef]
67. Ravindra, P.; Anyanwu, K. Nesting Strategies for Enabling Nimble MapReduce Dataflows for Large RDF Data. *Int. J. Semant. Web Inf. Syst.* **2014**, *10*, 1–26. [CrossRef]
68. Ravindra, P.; Anyanwu, K. Scaling Unbound-Property Queries on Big RDF Data Warehouses Using MapReduce. In Proceedings of the 18th International Conference on Extending Database Technology (EDBT), Brussels, Belgium, 23–27 March 2015; pp. 169–180.
69. Apache Pig. Available online: https://pig.apache.org/ (accessed on October 31 2015).
70. Choi, P.; Jung, J.; Lee, K.H. RDFChain: Chain Centric Storage for Scalable Join Processing of RDF Graphs Using MapReduce and HBase. In Proceedings of the 12th International Semantic Web Conference and the 1st Australasian Semantic Web Conference, Sydney, Australia, 21–25 October 2013; pp. 249–252.
71. Kim, H.S.; Ravindra, P.; Anyanwu, K. Scan-Sharing for Optimizing RDF Graph Pattern Matching on MapReduce. In Proceedings of the 2012 IEEE 5th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, 24–29 June 2012; pp. 139–146.
72. Ravindra, P.; Kim, H.S.; Anyanwu, K. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *The Semanic Web: Research and Applications*; Springer: Berlin Heidelberg, Germany, 2011; pp. 46–61.
73. Zhang, X.; Chen, L.; Wang, M. Towards Efficient Join Processing over Large RDF Graph Using MapReduce. In *Scientific and Statistical Database Management*; Springer: Berlin Heidelberg, Germany, 2012; pp. 250–259.
74. Apache Jena Core RDF API. Available online: http://jena.apache.org/documentation/rdf/index.html (accessed on October 31 2015).
75. Vitolo, C.; Elkhatib, Y.; Reusser, D.; Macleod, C.J.A.; Buytaert, W. Web Technologies for Environmental Big Data. *Environ. Model. Softw.* **2015**, *63*, 185–198. [CrossRef]
76. Jacob, F.; Johnson, A.; Javed, F.; Zhao, M.; McNair, M. WebScalding: A Framework for Big Data Web Services. In Proceedings of the IEEE First International Conference on Big Data Computing Service and Applications (BigDataService), Redwood City, CA, USA, 30 March–2 April 2015; pp. 493–498.
77. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking Cloud Serving Systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154.
78. Silberstein, A.; Sears, R.; Zhou, W.; Cooper, B.F. A Batch of PNUTS: Experiences Connecting Cloud Batch and Serving Systems. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, Athens, Greece, 12–16 June 2011; pp. 1101–1112.
79. Apache Spark. Available online: https://spark.apache.org/ (accessed on October 31 2015).
80. Abedjan, Z.; Grütze, T.; Jentzsch, A.; Naumann, F. Profiling and Mining RDF Data with ProLOD++. In Proceedings of the IEEE 30th International Conference on Data Engineering, Chicago, IL, USA, 30 March–4 April 2014; pp. 1198–1201.
81. Kushwaha, N.; Vyas, O.P. Leveragi0ng Bibliographic RDF Data for Keyword Prediction with Association Rule Mining (ARM). *Data Sci. J.* **2014**, *13*, 119–126. [CrossRef]

82. Cuzzocrea, A. A Framework for Modeling and Supporting Data Transformation Services over Data and Knowledge Grids with Real-Time Bound Constraints. *Concurr. Comput. Pract. Exp.* **2011**, *23*, 436–457. [CrossRef]
83. Cuzzocrea, A.; Saccà, D. Exploiting Compression and Approximation Paradigms for Effective And Efficient Online Analytical Processing over Sensor Network Readings in Data Grid Environments. *Concurr. Comput. Pract. Exp.* **2013**, *25*, 2016–2035. [CrossRef]