Distributed Keyword Search over RDF via MapReduce

Roberto De Virgilio and Antonio Maccioni

Dipartimento di Ingegneria
Universitá Roma Tre, Rome, Italy
{dvr,maccioni}@dia.uniroma3.it

Abstract

Non expert users need support to access linked data available on the Web. To this aim, keyword-based search is considered an essential feature of database systems. The distributed nature of the Semantic Web demands query processing techniques to evolve towards a scenario where data is scattered on distributed data stores. Existing approaches to keyword search cannot guarantee scalability in a distributed environment, because, at runtime, they are unaware of the location of the relevant data to the query and thus, they cannot optimize join tasks. In this paper, we illustrate a novel distributed approach to keyword search over RDF data that exploits the MapReduce paradigm by switching the problem from graph-parallel to data-parallel processing. Moreover, our framework is able to consider ranking during the building phase to return directly the best (top-k) answers in the first (k) generated results, reducing greatly the overall computational load and complexity. Finally, a comprehensive evaluation demonstrates that our approach exhibits very good efficiency guaranteeing high level of accuracy, especially with respect to state-of-the-art competitors.

Keywords: #eswc2014Virgilio

1 Introduction

The size of the Semantic Web is rapidly increasing due to numerous organizations that are opening up their databases on the Web following the linked data principles. This causes that, often, RDF datasets do not fit on a single machine. Moreover, data are linked only in a logical way, whereas, frequently, they are physically distributed over different locations. Obviously, RDF data management systems should be able to process distributed data [16]. There exists approaches to query distributed RDF data with SPARQL-like queries [8,13,17]. Usually, they exploit optimizations based on structural information (i.e. graph partitioning), but unfortunately, they cannot adapt to answer structure-free queries, such as keyword search-based queries. Keyword search is gathering the attention of Semantic Web practitioners, who want to support users in accessing linked (open) data. In fact, these users: (i) are usually unaware of the way in which data is organized, (ii) do not know how to interpret a Web ontology (if present) and (iii) do not know the syntax of a specific query language (e.g., SPARQL). Clearly, there is an imminent necessity to process efficiently keyword queries over distributed RDF data stores. Unfortunately, approaches for keyword search over RDF data

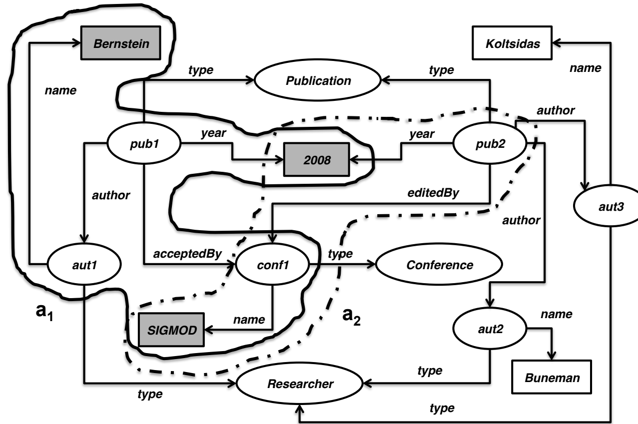

Fig. 1. An RDF graph G_1 from DBLP.

are all centralized (e.g., [6,15,18]). This is due to the fact that the keywords in the query do not help to identify how the RDF dataset has to be explored. Consequently, they are compelled to compute many expensive join operations over data distributed over different locations. Recently, MapReduce [5] has become a "de-facto" standard for distributed and parallel massive data processing on a cluster of commodity machines. MapReduce offers a high-level abstraction for solving problems through *rounds* of two independent and subsequent functions, i.e. *map* and *reduce*. MapReduce is an effective paradigm to compute algorithms that require to read the data just once and for this reason, it is not efficient to perform joins and graph algorithms [14].

Problem. Let us show with the example of G_1 in Fig. 1 how the RDF keyword queries are solved in order to point out why existing centralized approaches (i.e. [15]) are not feasible in this context. G_1 is a sample RDF version of the DBLP dataset (a database about scientific publications). Vertices in ovals represent entities, such as *aut1* and *aut2*, or concepts, such as *Conference* and *Publication*. Vertices in rectangles are literal values, such as *Bernstein* and *Buneman*. Edges describe connections between vertices. For instance, entity *aut1* is a *Researcher* whose name is *Bernstein*.

Given a keyword search query over RDF, a generic approach would: ① identify the vertices of the RDF graph holding the data matching the input keywords, ② traverse the edges to discover the connections between them to build n candidate answers (with $n > k$), and ③ rank answers according to a relevance criteria to return the top relevant k. While ① can be easily solved considering an indexing method over the content of the RDF graph, the task in ② requires to compute lot of joins over the distributed data, which are disk and network intensive. Moreover, the task in ② intrinsically computes more answers (an overset of the most relevant answers) than required and the task in ③ cannot be computed by independent parallel processes. It would be ideal to avoid ③ by generating exactly the best k answers in ②. Considering our running example with the query $Q_1 = \{$`Bernstein, SIGMOD, 2008`$\}$, a_1 (i.e. articles of *Bernstein*

published in *SIGMOD 2008*) is intuitively more relevant than a_2 (i.e. articles of *Buneman* published in *SIGMOD 2008*) because it includes more keywords and it should be retrieved as the first answer. Note that ranking functions consider more elaborate criteria to evaluate the relevance of an answer. Ideally, if one is interested in the top-2 answers, only a_1 and a_2 shall be computed, hence avoiding a ranking procedure. In this way, at each step the answer generated is the best possible in the sense that the answers generated next cannot have a higher relevance. This property is called *monotonicity* and a process satisfying such property is a *monotonic* process. A monotonic process allows to reduce the *time-to-result* because the user can get an answer before the computation of the following one is completed. Hence, a monotonic processing is particularly desirable in the context of MapReduce to mitigate the latency of the execution.

Contribution. In this paper, we present a novel distributed keyword-based search technique over RDF data that builds the best k results in the first k generated answers. We exploit the MapReduce [5] paradigm in order to benefit from the features (i.e. load balancing, fault tolerance, job scheduling, etc.) that current implementations (e.g., Hadoop or variants thereof) give to developers. Nevertheless, MapReduce is a data-parallel paradigm that is not very efficient for the processing of RDF data, which usually requires join-intensive tasks. By exploiting a path-store for RDF, we reverse the distributed keyword search over RDF from a graph-parallel problem to a data-parallel problem. This store is an implementation of the work in [1] on Hadoop/HBase and it is not a contribution for this paper. Intuitively, in this store, the connections among elements (i.e. adjacencies and intersections) of the RDF graph are explicitly stored in *paths*, allowing us to avoid, in this way, the computation of joins. In general, while existing approaches explore the dataset to find sub-graphs holding relevant information to the query, we only combine the RDF paths according to their precomputed intersections. In addition, since the relevance of answers is highly dependent on both the construction of candidates and on their ranking, our query answering combines search and ranking. In our approach the RDF paths that are relevant to the query are retrieved from the store and then grouped (i.e. clustered) with respect to a criteria that captures the type of information in the path. Then, we compute, at each step, the most relevant answer by picking and combining the most relevant paths from each group. The relevance is given by a scoring function. Note that, however, our approach is scoring functions agnostic. Note that this approach is inspired by the theoretical results in [4], but it proposes different algorithms. To validate our approach, we have developed a distributed system for keyword-based search over RDF data that implements the techniques described in this paper. Experiments over widely used benchmarks have shown very good results with respect to other approaches, in terms of both effectiveness and efficiency.

Outline. The rest of the paper is organized as follows. Section 2 introduces some preliminary issues. Section 3 overviews the proposed approach to keyword search, while Section 4 illustrates the distributed algorithms in detail. In Section 5, we

discuss related research and in Section 6, we present the experimental results. Finally, in Section 7, we draw our conclusions and sketch future research.

2 Basic Concepts

This section introduces some preliminary notions and the problem we address.

Data Structures. RDF datasets are naturally represented as labeled directed graphs.

Definition 1 (RDF Data Graph). *An RDF data graph is a labeled directed graph* $G = \{V, E, \Sigma_V, \Sigma_E, L_G\}$ *where* V *is a set of vertices and* $E \subseteq V \times V$ *is a set of ordered pairs of vertices, called edges.* Σ_V *and* Σ_E *are the sets of vertices and edge labels, respectively. The labeling function* L_G *associates an element of* V *to an element of* Σ_V *and an element of* E *to an element of* Σ_E*.*

We call *start* vertex, the vertices of G with no in-going edges, and *end* vertices, the vertices of G with no out-going edges. Basically, a *path* in a graph G is a sequence of labels from a start vertex to an end vertex of G. In the case of cycles, a path ends, intuitively, just before the repetition of a vertex label. Moreover, if there is no start vertex in G, a path starts from vertices whose difference between the number of outgoing edges and the number of the incoming edges is maximal in G. We call these vertices *hubs*.

Definition 2 (Path). *Given a graph* $G = \{V, E, \Sigma_V, \Sigma_E, L_G\}$*, a path is a sequence* $p = l_{v_1} - l_{e_1} - l_{v_2} - l_{e_2} - \ldots - l_{e_{v-1}} - l_{v_j}$ *where: (i)* $l_{v_i} = L_G(v_i)$*,* $l_{e_i} = L_G(e_i)$*, and* $v_i \in V$*,* $e_i = (v_i, v_{i+1}) \in E$*, (ii)* v_1 *is either a start vertex or, if* G *has no start vertices, a hub, and (iii)* v_j *is either an end vertex or a vertex such that there is no edge* (v_j, v_{j+1}) *such that* $L_G(v_{j+1})$ *(the label of* v_j*) already occurs in* p*.*

In the following, we will call v_1 and v_j the *source* and the *sink* of a p, respectively. The *length* of a path is the number of vertices occurring in the path, while the *position* of a vertex corresponds to its position among the vertices in the path. For instance, the graph in Fig. 1 has two sources: *pub1* and *pub2*. An example of path is $p_2 =$ pub1-author-aut1-name-Bernstein, whose length is 3 and the vertex aut1 has position 2. In our approach we exploit a path-store inspired by [1] that indexes all paths starting from a source and ending with a sink. Obviously, at running time we are interested in the paths relevant to the query, that is, the paths whose sinks match at least one keyword of the query. As in [15], we assume that users enter keywords corresponding to attribute values, that are necessarily within the sink's labels. This is not a limitation because vertices labeled by URIs are usually linked to literals, which represent verbose descriptions of such URIs. In our implementation, the operation of *matching* is executed with standard libraries based on full text search techniques (such as stemming). Since this aspect is outside the scope of the paper, we will simply assume, hereinafter, that the operation of matching provides a support for imprecise matching between labels and we will use the term matching between values in this sense,

without discussing this aspect further. In a path, the sequence of edge labels describes the corresponding structure. To some extent, such a structure describes a schema for the values on vertices that share the same connection type. While we cannot advocate the presence of a schema, we can say that such a sequence is a *template* for the path. Therefore, given a path p, its template t_p results from the path where each vertex label is replaced with the wild-card #. In the example of Fig. 1, the template t_{p_2} associated to p_2 = pub1-author-aut1-name-Bernstein is #-author-#-name-#. We say that p_2 *satisfies* t_{p_2}, denoted with $p_2 \approx t_{p_2}$. Multiple paths that share the same template can be considered as homogeneous.

When two paths p_i and p_j share a common vertex, we say that there is an intersection between p_i and p_j and we indicate it with $p_i \leftrightarrow p_j$. Finally, an answer a to Q over G is a set of paths forming a connected components, i.e. a directed labeled sub-graph of G where the paths present pairwise intersections as defined below.

Definition 3 (Answer). *An answer a is a set of paths $p_1, p_2, \ldots, p_n$ where $\forall p_a, p_b \in a$ there exists a sequence $[p_a, p_{w_1}, \ldots, p_{w_m}, p_b]$, with $m < n$, such that $p_{w_i} \in a$, $p_a \leftrightarrow p_{w_1}$, $p_b \leftrightarrow p_{w_m}$, and $\forall i \in [1, m\text{-}1] : p_{w_i} \leftrightarrow p_{w_{i+1}}$.*

Note that, since the paths form a connected component, our notion of answer basically corresponds to the notion of RDF sub-graph answer used by other approaches.

Ranking and monotonicity. To assess the relevance of an answer a for a query Q, a scoring function $score(a, Q)$ is adopted. It returns a number that is greater when the answer is more relevant. Then, the ranking is given by ordering the answers according to their relevance. A query answering process is monotonic if the i-th generated answer is always more relevant than the $(i+1)$-th. It follows that, if a query answering is monotonic, a ranking task is needless. Since a single path can be an answer, it is reasonable to consider the same scoring function to evaluate both paths and answers. In the following sections, we will use the notation $score(p, Q)$ and $score(a, Q)$ to evaluate the relevance of a path p and of an answer a with respect to the query Q, respectively. We remark that, unlike all current approaches, we are independent from the scoring function: we do not impose a monotonic, aggregative nor an "ad-hoc for the case" scoring function.

Problem definition. Given a labeled directed graph G and a keyword search based query $Q = \{q_1, q_2, \ldots, q_n\}$, where each q_i is a keyword, we aim at finding the top-k ranked answers $a_1, a_2, \ldots, a_k$ to Q.

3 Monotonic Keyword Search

This section overviews our distributed approach to keyword search over RDF and discusses the conditions under which the answer generation process exhibits a monotonic behaviour. The flow of execution is shown in Fig. 2, but the algorithms of each block will be detailed in the next Section 4.

Let G be an RDF data graph and Q a keyword query over it. Our approach provides two main phases: the *indexing* (done off-line), in which all the paths of

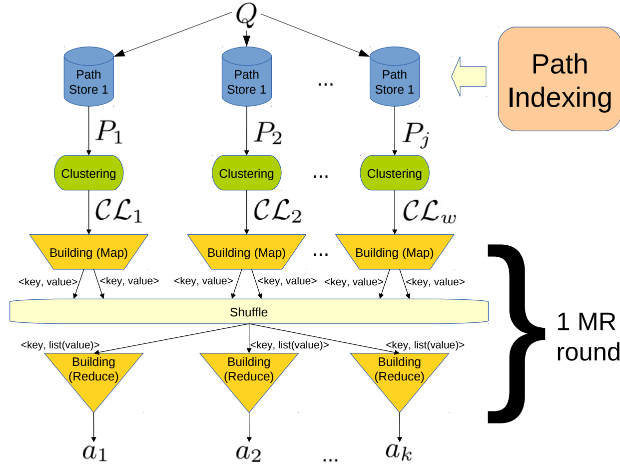

Fig. 2. The flow of execution.

G are indexed in the store (the orange part of Fig. 2), and the *query processing* (done on-the-fly), where the query evaluation takes place. The first task will be described in more detail in Section 6. Intuitively, let us consider a distributed environment composed by two machines. For example, the paths of G can be stored within the two independent path stores in Fig. 3. Note that the replication of data is transparent to the developer in MapReduce, so we assume the content of the two path stores to be disjoint.

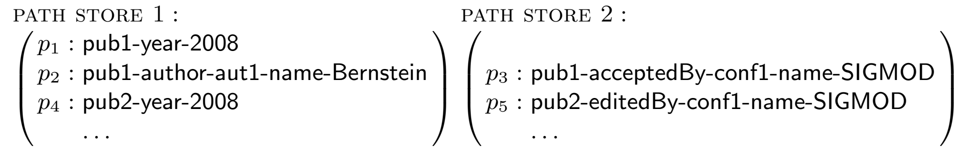

Fig. 3. Distributed Path-Store.

In the second phase, all paths P relevant for Q (i.e. all paths whose sinks match at least one keyword of Q) are retrieved by exploiting the store. Then, the best answers are generated from P (blue part of Fig. 2). An important feature of this phase is the use of the scoring function while computing the answers. This phase is performed by the following two main tasks:

Clustering. In this task (the green part of Fig. 2) we group the paths of P into clusters according to their template. Every distributed node manages a set of clusters, that in our running example with Q_1 over G_1 are $\mathcal{CL}_1$ and $\mathcal{CL}_2$ as showed in Fig. 4. In this case clusters cl_1, cl_2, cl_3 and cl_4 correspond to the different templates extracted from P. Before the insertion of a path p in the cluster, we evaluate its score. The paths in a clusters are ordered according to their score with the greater coming first, i.e. $score(p_1, Q_1) \geq score(p_4, Q_1)$. It is straightforward to demonstrate that the time complexity of the clustering is $O(|P|)$: it executes $|P|$ insertions into the clusters.

$\mathcal{CL}_1$:
$\begin{pmatrix} cl_1[\text{\#-year-\#}] : & p_1, p_4 \\ cl_2[\text{\#-author-\#-name-\#}] : & p_2 \end{pmatrix}$

$\mathcal{CL}_2$:
$\begin{pmatrix} cl_3[\text{\#-acceptedBy-\#-name-\#}] : p_3 \\ cl_4[\text{\#-editedBy-\#-name-\#}] : p_5 \end{pmatrix}$

Fig. 4. Clustering of paths.

Building. The last task aims at generating the most relevant answers by combining the paths in the clusters (yellow part of Fig. 2). This is done by picking and combining the paths with greatest score from each cluster, i.e. the most promising paths. Note that we diversify the answer content by not including homogeneous data, that is paths from the same cluster.

The combination of paths is led by a strategy that decides whether a path has to be inserted in a final answer or not. Two different strategies are defined:

1. *linear strategy*: it guarantees a linear time complexity with respect to the size of the input in a single round of MapReduce. Basically, the final answers are the connected components of the most relevant paths of the clusters.
2. *monotonic strategy*: it generates the answers in order according to their relevance in a quadratic time complexity with respect to the size of the input. It completes in $2 \times k$ rounds of MapReduce, at most. As the linear strategy, it computes the connected components from the most relevant paths in the clusters, but differently, the path interconnection is not the only criterion to form an answer. At this point every connected components is analysed to check if it fulfils the monotonicity, that is to check if the answer we are generating is optimum. This check is supported by the so called τ-*test*, which is explained in [4].

Algorithm 1: LinearBuilding (Map)

Input : The map $\mathcal{CL}$.
Output: A list of pairs $< key, value >$.

```
1 iteration ← 1;
2 while CL is not empty do
3     foreach cl ∈ CL do
4         first_cl ← cl.DequeueTop() ;
5         foreach p ∈ first_cl do
6             output(<iteration, p >) ;
7     iteration ← iteration + 1;
```

4 MapReduce Building Strategies

Given the query Q and the set P of paths matching the query Q, retrieved from the path-store, we compose those paths to generate the final top-k answers. In the following we discuss separately the two strategies implemented in MapReduce.

4.1 The Linear Strategy

Given the set of clusters $\mathcal{CL}$, the building of answers is performed by generating the connected components `CC` from the most promising paths in $\mathcal{CL}$. The linear strategy requires only one round of MapReduce. Every machine executes a *map* job (Algorithm 1), while the total number of *reduce* jobs is the maximum number of different scores present in the clusters. Algorithm 1 iterates over the local clusters (lines [3-6]) to extract the top paths from `cl` (line [4]), i.e. all paths having the same (top) score. The iterations continue until the clusters are empty (line [2]). At each iteration, the top paths are inserted in `first_cl` and each of them is returned as *value* in a pair with the number of the iteration as *key* (line [6]).

Algorithm 2: LinearBuilding (Reduce)

Input : A pair $< key, value >$.
Output: A list of answers.

```
1 CC ← FindCC(value);
2 ans ← ∅;
3 foreach cc ∈ CC do
4     ans.Enqueue(cc);
5 return ans ;
```

A reduce function (Algorithm 2) receives a list *value* of paths extracted during the same iteration and an integer *key* (that is not used). Out of *value*, we compute the connected components `CC` (line [1]), each of which represents an answer. Referring again to our example, the mappers produce the pairs $< 1, p_1 >$, $< 1, p_2 >$, $< 1, p_3 >$, $< 1, p_5 >$ and $< 2, p_4 >$. After the shuffle phase, we have the pairs $< 1, (p_1, p_2, p_3, p_5) >$ and $< 2, (p_4) >$ which are assigned to different reducers. The first reducer produces $cc_1 = \{p_1, p_2, p_3, p_5\}$, while the other produces $cc_2 = \{p_4\}$. All the answers are returned in output (line [5]). Note that, if we retrieve k answers, we stop the launch of more reducers.

Computational Complexity. Algorithm 1 and Algorithm 2 produce the answers in linear time with respect to the number I of paths matching the input query Q. The Algorithm 1 is in $O(I)$ because it makes I extractions and I insertions of pairs. Algorithm 2 is in $O(I)$ in the worst case, which happens when only one reducer is called (the case is similar to a sequential version of the algorithm). Therefore, the computation is given by the call of `FindCC` that has a complexity of $O(I)$ (it iterates over I paths once, see pseudo-code in [4]) followed by I insertions at most. We can state that the overall linear strategy is in $O(I)$.

Ranking. This strategy produces good answers efficiently; in our example, we have $a_1 = \{p_1, p_2, p_3, p_5\}$ and $a_2 = \{p_4\}$. Observing the answers with respect to Q_1, a_1 contains the unnecessary p_5, while a_2 is partially incomplete (i.e. it should include p_5). Such strategy tends to produce *exhaustive* answers but not optimally *specific*, that is to include all relevant information matching the query but not optimally limiting the irrelevant ones. The answer generated at each step may not

be the optimum answer: it may happen to generate a sequence of two answers, a_i and a_{i+1}, where $score(a_{i+1}, Q) > score(a_i, Q)$. Moreover, since we output the answers as soon as they are built, we do not control if a_{i+1} is completed before a_i. The ranking of this strategy cannot guarantee monotonicity.

4.2 The Monotonic Strategy

In this section, we provide a second strategy that implements the *τ-test* (see Theorem in [4]) while building the answers. In this case the iterations are dependent on each other because discarded paths have to be used in the next iterations.

Algorithm 3: MonotonicBuilding (Map Round 1)

Input : The map $\mathcal{CL}$.
Output: A list of pairs $< key, value >$.

```
1 first ← ∅;
2 foreach cl ∈ CL do
3     first ← first ∪ cl.DequeueTop() ;
4 CC ← FindCC(first);
5 foreach cc ∈ CC do
6     output(<1, cc>) ;
```

To produce an answer we launch two rounds of MapReduce. The first round (Algorithm 3 and Algorithm 4) produces the connected components among the most relevant distributed paths; the second round (Algorithm 5 and Algorithm 6) analyses the connected components in parallel, producing the final answers. After the clustering of paths in $\mathcal{CL}$, the mappers (Algorithm 3) compute the local connected components (line [4]) of the most relevant paths (lines [2-3]). Then, each connected component *cc* is sent to the same reducer (lines [5-6]). The reducer of the first round is in Algorithm 4. It takes all the locally computed connected components and produce the global connected components. It calls the function `FindCC` (line [1]) taking advantage from partially computed connected components of the input.

Algorithm 4: MonotonicBuilding (Reduce Round 1)

Input : A pair $< key, value >$.
Output: a list of global connected components.

```
1 CC ← FindCC(value);
2 return CC ;
```

The map function of the second round (Algorithm 5) dispatches every global connected component *cc* to a different reducer (lines [2-4]).

Every reducer of the second round (Algorithm 6) receives a connected component as *value* and launches a local analysis procedure, i.e. `MonotonicityAnalysis` (line [1]), to apply the *τ-test*. For the purpose, the reducer uses as input the more relevant path p_s in $\mathcal{CL}$. This path is kept as a global

Algorithm 5: MonotonicBuilding (Map Round 2)

Input : The list of connected components CC.
Output: A list of pairs $< key, value >$.

```
1 i ← 0;
2 foreach cc ∈ CC do
3     i ← i + 1;
4     output(<i, cc>) ;
```

variable within the distributed environment[1] . At the end of the analysis, an optimal answer a is returned (line [3]) and the discarded paths of the connected components *value* are inserted back into the clusters (line [2]). Note that, since there are more reducers returning an optimum answer, we only output one of them to the user and the others are reinserted into the clusters.

Algorithm 6: MonotonicBuilding (Reduce Round 2)

Input : A pair $< key, value >$, a path p_s.
Output: an answer.

```
1 a ← MonotonicityAnalysis(value, ∅, p_s);
2 InsertPathsInClusters(value, CL);
3 return a;
```

Monotonicity Analysis. Algorithm 7 checks if the answer we are generating is (still) optimum, thus, it preserves the monotonicity. It is a recursive function that generates the set `OptAns` of all answers (candidate to be optimum) by combining the paths in a connected component *cc*. At the end, it returns an answer `optA` given by the maximal and optimum subset of paths in *cc*. It takes as input the connected component *cc*, the current optimum answer `optA` and the top path p_s contained in $\mathcal{CL}$. If *cc* is empty, we return `optA` as it is (lines [1-2]). Otherwise, we analyze all paths $p_x \in cc$ that present an intersection with a path p_i of `optA` ($p_x \leftrightarrow p_i$). If there is no intersection, then `optA` is the final optimum answer (lines [6-7]). Otherwise, for each p_x, we calculate τ (line [10]), through the function `getTau`, and then execute the *τ-test* on each new answer `optA'`, that is `optA` $\cup \{p_x\}$. If `optA'` satisfies the *τ-test* (line [11]), then it represents the new optimum answer: we insert it into `OptAns` and we invoke the recursion on `optA'` (line [12]). Otherwise, we keep `optA` as optimum answer and skip p_x (line [14]). At the finish, we want an optimal answer that is not a subset of any other. This is done by selecting the maximal `optA` from `OptAns` by using `TakeMaximal` (line [15]).

Let us consider our running example. In the first round we produce the connected components. The mappers produce the local connected components through the pairs $< 1, \{p_1, p_2\} >$ and $1, \{p_3, p_5\} >$, the reducer produces one

[1] The global variable is implemented by means of the Hadoop's Distributed Cache functionality

global connected component $cc_1 = \{p_1, p_2, p_3, p_5\}$, whose paths have, by using the scoring function implemented in [4], score 2.05, 1.63, 1.6 and 1.49, respectively. In the reduce phase of the second round, we analyse all possible combinations of the paths in cc_1 to find the optimum answer(s). Therefore, at the beginning we have `optA` $= \{p_1\}$, since p_1 has the highest score, and p_s is p_4 (i.e. it is the only path in the clusters). The value of τ is 1.86. Then, the algorithm retrieves the following admissible optima answers: $a'_1 = \{p_1, p_2, p_3\}$, $a'_2 = \{p_1, p_3\}$, and $a'_3 = \{p_1, p_2, p_5\}$. These answers are admissible because they satisfy the τ-*test* and their paths present pairwise intersections. During computation, the analysis skips answers $a'_4 = \{p_1, p_2, p_3, p_5\}$ and $a'_5 = \{p_1, p_3, p_5\}$ because they do not satisfy the τ-*test*: they have scores 1.55 and 1.26, respectively. Finally, the function `TakeMaximal` selects a'_1 as the final first optimum answer a_1 since it has more paths and the highest score. Following a similar process, at the second round, the algorithm returns $a_2 = \{p_4, p_5\}$ with a lesser score than a_1.

Algorithm 7: Monotonicity Analysis

Input : A set of paths cc, an answer optA, a path p_s.
Output: The new (in case) optimum answer optA.

```
1  if cc is empty then
2  |  return optA;
3  else
4  |  OptAns ← ∅;
5  |  foreach p_x ∈ cc do
6  |  |  if (∄p_i ∈ optA : p_x ↔ p_i) and optA is not empty then
7  |  |  |  OptAns ← OptAns ∪ optA ;
8  |  |  else
9  |  |  |  optA' ← optA ∪ {p_x};
10 |  |  |  τ ← getTau(cc - {p_x }, p_s);
11 |  |  |  if score(optA', Q) ≥ τ then
12 |  |  |  |  OptAns ← OptAns ∪ MonotonicityAnalysis(cc - {p_x }, optA',
   |  |  |  |  p_s);
13 |  |  |  else
14 |  |  |  |  OptAns ← OptAns ∪ optA ;
15 |  optA ← TakeMaximal(OptAns) ;
16 |  return optA;
```

Computational Complexity. Following the discussion illustrated in [4], although this analysis achieves our goal, the computational complexity of the generation process is in $O(I^2)$, where I is the number of paths matching the input query Q. The execution of the first round functions is in $O(I)$ because: in the mapper we have that lines [2-3] are in $O(|\mathcal{CL}|) \in O(I)$, line [4] (it iterates over I paths once, see pseudo-code in [4]) and lines [5-6] are also in $O(I)$; line [1] of the reducer is in $O(I)$. Then, the connected components $cc \in$ `CC` are parallelly analysed in the second round. Algorithm 5 iterates over `CC` and therefore it is in $O(I)$. Algorithm 6 computes at most I insertions (through `InsertPathsInClusters`)

and launches the function `MonotonicityAnalysis` that is in $O(I^2)$. This is a recursive function where the main executions are in lines [9-12] and line [15]. Both the executions are in $O(I)$, since we have I elements to analyse at the most. The recursion is called at most I times and therefore `MonotonicityAnalysis` is in $O(I^2)$, which is also the computational time complexity of the overall monotonic strategy. In the worst case, the computation is computed in $2 \times k$ rounds, two for each generated answer. A comprehensive discussion about the quality and accuracy of the answers according to measures of *exhaustivity* ($\mathcal{EX}$) and *specificity* ($\mathcal{SP}$) can be found in [4].

5 Related Work

We consider two different categories of related work that we discuss separately in the following.

RDF Keyword Search. Ad-hoc approaches for RDF have been proposed [6,15,18]. The work in [15] proposes a semi-automatic system to interpret the query into a set of candidate conjunctive queries. Users can refine the search by selecting the computed candidate queries to submit that represent their information need. Candidate queries are computed exploring the top-k sub-graphs matching the keywords. The approach in [18] relies on a RDFS domain knowledge to convert keywords in query-guides, which help users to incrementally build the desired semantic query. While unnecessary queries are not built (thus not executed), there is a strict dependency on user feedback. The work in [6] employs a ranking model based on IR and statistical methods.

Distributed RDF Processing. The distributed nature of the Semantic Web infrastructure arises the necessity to compute RDF data in a parallel and distributed fashion. Most of the works in this context [10,7,9,8,11,13,17] employ a distributed environment of RDF data-stores and compute SPARQL queries over them. In particular, the works in [10,8,9,11] make use of Hadoop or Hadoop/HBase to compute joins and manage the distribution of the query. DARQ [13] implements a framework to solve federated SPARQL queries. The work in [16] proposes a scalable P2P system for computing reasoning on RDF(S), while the work in [12] solves RDF path queries in MapReduce. All of these approaches exploit the structural information of the RDF data graph and are not applicable for solving keyword search queries, which miss such specifications. For this reason, all existing RDF keyword search approaches are centralized. TrinityRDF [17] considerably differs from the other approaches. It is an in-memory store where data is natively modelled as a graph. TrinityRDF is able to expedite the query processing by optimizing graph random accesses. Unfortunately, it requires a huge amount of memory and it is more suitable for a cloud infrastructure rather than a cluster of commodity machines.

6 Experimental Evaluation

We implemented our approach in YaaniiMR, a Java system for keyword search over RDF graphs, that is a MapReduce implementation of Yaanii discussed in [4]. In our experiments, we used the benchmark provided by Coffman et al. [2] that employs two well-know datasets, IMDb and Wikipedia, and an ideal counterpart due to its smaller size, Mondial. In our context, we used the RDF versions of three datasets: DBPedia (i.e. 266M triples) including *Linked IMDb*[2] for the benchmark related to IMDb, Billion (i.e. 1000M triples) that is the *Billion Triple Challenge Dataset 2008*[3] including *Wikipedia3*[4] for Wikipedia, while for Mondial we converted the SQL dump into RDF ourselves (i.e. 15000 triples). For each dataset, we run the set of 50 queries provided by [2] (see the paper for details and statistics). Since the results, and therefore the effectiveness, of YaaniiMR is the same of Yaanii [4], in this section we focus exclusively on the performance.

Benchmark Environment. We deployed YaaniiMR on EC2 clusters. In particular, to fully understand the scale-up properties of YaaniiMR, we consider three EC2 *clusters quadruple* (i.e. *cc1.4xlarge* configuration as detailed by Amazon Web Service): 10, 50 and 100 nodes. YaaniiMR is provided with the Hadoop file system (HDFS) version 1.1.1 and the HBase data store version 0.94.3, and compiled and run using Java 7. The performance of our systems has been measured with respect to data loading, memory footprint, and query execution.

Data Loading. We employ a path-based store on RDF for a distributed environment that is the evolution of the index in [1]. This is stored in HBase and supported by the distributed file system HDFS. The RDF dataset is indexed off-line. In particular, we index all the shortest paths starting from a source and ending with a sink. This task is efficiently performed by an optimized implementation of the *Breadth-first search* (BFS) strategy through MapReduce [3]. At running time we use the index to retrieve the paths whose sink node matches a keyword.

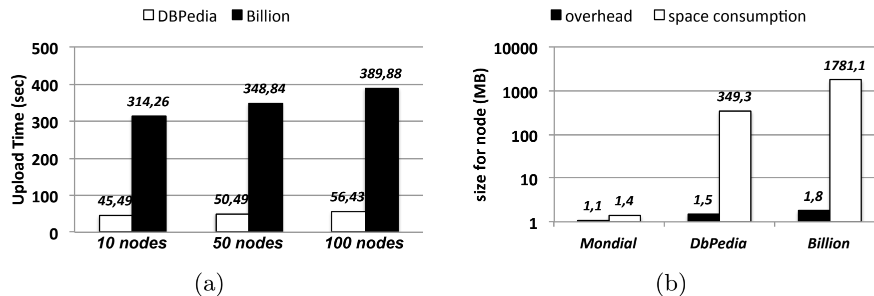

Fig. 5. (a) Data loading times in seconds and (b) Data Space consumption expressed in MB in 10-nodes EC2 cluster.

[2] Available at http://linkedimdb.org/

[3] Available at http://challenge.semanticweb.org/

[4] Available at http://stats.lod2.eu/rdfdocs/228

We used the three EC2 clusters to evaluate the upload time (the time to build all paths and to import them into our index) as showed in Fig. 5.(a). In particular the figure illustrates times only for DBPEDIA and BILLION, since MONDIAL spends only few seconds for starting up all jobs. Both DBPEDIA and BILLION achieve roughly the same upload times in any configuration, providing a linear trend: overall, these results show the efficiency of data loading in our system when scaling-out. Another significant advantage of our system relies in memory and space consumption. Let us consider the 10-nodes EC2 cluster. The overall memory overhead needed to maintain our index remains almost constant, and amounts to circa 1 MB of RAM for node. The total distributed space consumption for node scales perfectly with the size of the dataset. As illustrated in Fig. 5.(b), we have 1.4 MB, 349.3 MB and 1781.1 MB for node to store MONDIAL, DBPEDIA and BILLION in our cluster. Proportionally, we have the same behaviour for 50 and 100 nodes.

MapReduce Job Execution. The second evaluation analyses the performance of YAANIIMR when running MapReduce jobs. To this aim we measure the *end-to-end* job run-times, which is the time a given job takes to run completely. In this case, we perform three runs for each of the 50-queries (over the three datasets) to retrieve the top-10 answers: at the end we evaluate the geometric mean of all the runs of the 50 queries for dataset in the three EC2 clusters configurations (i.e. 10, 50 and 100 nodes), as shown in Fig. 6. In the figure, the job runtime is made by the *ideal time* (T_{ideal}) and the *overhead* ($T_{overhead}$) that are the effective time to run the query and the time to startup all necessary jobs, respectively. For each dataset we evaluate the average response time (i.e. computed by the geometric mean of times) of all queries executed by using both the linear, i.e. **L**, and the monotonic, i.e. **M**, strategy. Finally we replicated this experiment in 10-nodes, 50-nodes and 100-nodes cluster.

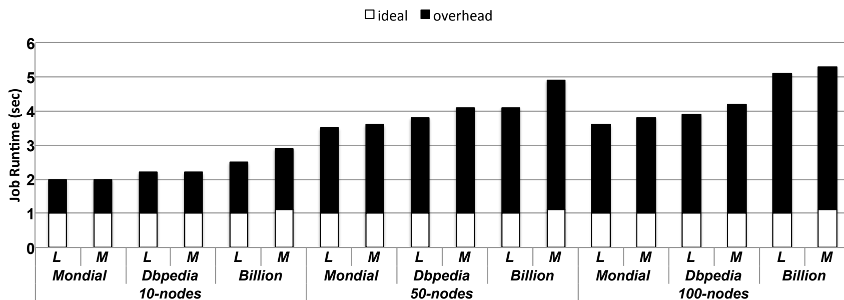

Fig. 6. End-to-end job runtimes

As we can see, $T_{overhead}$ dominates the total job runtime. To schedule a single job, Hadoop spends 1 or 2 seconds even though the actual task (query execution) just runs in a few ms. Therefore, it is comparable to YAANII [4]. Our implementation in MapReduce scales perfectly with respect to both the number of machines (i.e. 10, 50 and 100) and the dataset size (i.e. MONDIAL, DBPEDIA and BILLION).

7 Conclusions and Future Work

In this paper, we presented a novel approach to distributed keyword search query over large RDF datasets. It relies on a MapReduce environment and comprise two strategies for top-k query answering. The linear strategy enables the search to scale seamlessly with the size of the input, while the monotonic strategy guarantees the monotonicity of the output. Experimental results confirmed our algorithms and the advantage over other approaches. This work now opens several directions of further research. From a practical point of view, we are widening a more synthetic catalogue to store information and optimization techniques to speed-up the index creation and update.

References

1. Cappellari, P., De Virgilio, R., Maccioni, A., Roantree, M.: A path-oriented rdf index for keyword search query processing. In: DEXA. pp. 366–380 (2011)
2. Coffman, J., Weaver, A.: An empirical performance evaluation of relational keyword search techniques. TKDE 99(PrePrints), 1 (2012)
3. Cosulschi, M., Cuzzocrea, A., De Virgilio, R.: Implementing bfs-based traversals of rdf graphs over mapreduce efficiently. In: CCGRID. pp. 569–574 (2013)
4. De Virgilio, R., Maccioni, A., Cappellari, P.: A linear and monotonic strategy to keyword search over rdf data. In: ICWE. pp. 338–353 (2013)
5. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI. pp. 137–150 (2004)
6. Elbassuoni, S., Blanco, R.: Keyword search over rdf graphs. In: CIKM (2011)
7. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered rdf store. In: SSWS (2009)
8. Huang, J., Abadi, D.J., Ren, K.: Scalable sparql querying of large rdf graphs. PVLDB 4(11), 1123–1134 (2011)
9. Husain, M.F.: Heuristics-based query processing for large rdf graphs using cloud computing. TKDE 23(9), 1312–1327 (2011)
10. Husain, M.F., Doshi, P., Khan, L., Thuraisingham, B.M.: Storage and retrieval of large rdf graph using hadoop and mapreduce. In: CloudCom. pp. 680–686 (2009)
11. Papailiou, N., Konstantinou, I., Tsoumakos, D., Koziris, N.: H2rdf: adaptive query processing on rdf data in the cloud. In: WWW. pp. 397–400 (2012)
12. Przyjaciel-Zablocki, M., Schätzle, A., Hornung, T., Lausen, G.: Rdfpath: Path query processing on large rdf graphs with mapreduce. In: ESWC Workshops. pp. 50–64 (2011)
13. Quilitz, B., Leser, U.: Querying distributed rdf data sources with sparql. In: ESWC. pp. 524–538 (2008)
14. Ravindra, P., Hong, S., Kim, H., Anyanwu, K.: Efficient processing of rdf graph pattern matching on mapreduce platforms. In: DataCloud-SC '11. pp. 13–20 (2011)
15. Tran, T., Wang, H., Rudolph, S., Cimiano, P.: Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In: ICDE. pp. 405–416 (2009)
16. Tsatsanifos, G., Sacharidis, D., Sellis, T.K.: On enhancing scalability for distributed rdf/s stores. In: EDBT. pp. 141–152 (2011)
17. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale rdf data. PVLDB 6(4), 265–276 (2013)

18. Zenz, G., Zhou, X., Minack, E., Siberski, W., Nejdl, W.: From keywords to semantic queries - incremental query construction on the semantic web. Journal of Web Semantics 7(3), 166–176 (2009)