A Linear and Monotonic Strategy to Keyword Search over RDF Data

Roberto De Virgilio[1], Antonio Maccioni[1], Paolo Cappellari[2]

[1]Università Roma Tre, Rome, Italy
{dvr,maccioni}@dia.uniroma3.it
[2]Dublin City University, Dublin, Ireland
pcappellari@computing.dcu.ie

Abstract

Keyword-based search over (semi)structured data is today considered an essential feature of modern information management systems and has become an hot topic in database research and development. Most of the recent approaches to this problem refer to a general scenario where: (i) the data source is represented as a graph, (ii) answers to queries are sub-graphs of the source containing keywords from queries, and (iii) solutions are ranked according to a relevance criteria. In this paper, we illustrate a novel approach to keyword search over semantic data that combines a solution building algorithm and a ranking technique to generate the best results in the first answers generated. We show that our approach is monotonic and has a linear computational complexity, greatly reducing the complexity of the overall process. Finally, experiments demonstrate that our approach exhibits very good efficiency and effectiveness, especially with respect to competing approaches.

1 Introduction

The amount of data in the Semantic Web is exponentially increasing due to organizations that are opening up data in the form of *linked data* and, on the other side, to users that are interested in using them. In general, to access (semantic) data users must know how data is organized (e.g., Web ontologies) and the syntax of a specific query language (e.g., SPARQL). Clearly, this is an obstacle to information access and retrieval. For this reason Keyword Search (KS) systems are increasingly capturing the attention of researchers and industry, since they provide an effective facilitation to non-expert users. Let us consider the example in Fig. 1. Graph 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* of name *Bernstein*. Typically, given a keyword search query, a generic approach would: i) identify the vertices of the RDF graph holding the data matching the input keywords, ii) traverse the edges to discover the connections (i.e. trees or sub-graphs) between them that build n candidate solutions (with n > k), and iii) rank solutions according to a relevance criteria to return the top relevant k. Intrinsically, this

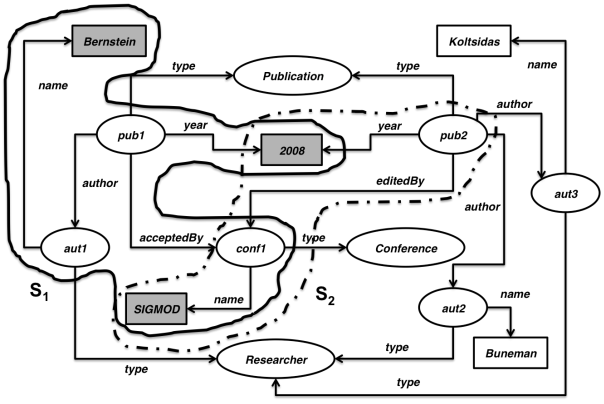


Fig. 1. An RDF graph G_1 from DBLP.

process generates (or computes) more solutions than required (an overset of the most relevant answers), whereas it would be ideal to generate exactly the best k. For instance, if one is interested in the top-2 answers for the query Q_1 = {`Bernstein, SIGMOD, 2008`} over G_1 in Fig. 1, then only S_1 (i.e. articles of *Bernstein* published in *SIGMOD 2008*) and S_2 (i.e. articles of *Buneman* published in *SIGMOD 2008*) shall be computed. Intuitively, S_1 is more relevant than S_2 because it includes more keywords and it should be retrieved as the first answer. Note that ranking functions consider more elaborated criteria to evaluate the relevance of an answer. It turns out however, that the relevance of answers is highly dependent on both the construction of candidates and their ranking. For this reason, the tasks of searching and of ranking are strongly correlated.

In this paper, we present a novel keyword based search technique over RDF graph-shaped data that builds the best k results in the first k solutions generated. This technique is inspired by a previous work [4]. The work in [4] builds top-k solutions in an approximate and sequential way focusing exclusively on the quality of the results. Differently, in this paper we address efficiency and scalability, beyond effectiveness, providing new algorithms to optimize the complexity of finding the best answers. To validate our approach, we have developed a system for keyword-based search over RDF data that implements the techniques described in this paper. Experiments over widely used benchmarks (Coffman et al. [3]) shows very good results with respect to other approaches, in terms of both effectiveness and efficiency. Specifically, we propose two different strategies for our framework. The first presents a linear computational cost and enables the search to scale seamlessly with the size of the input. The second, inspired by the Threshold Algorithm proposed by Fagin et al. [6], guarantees the monotonicity of the output as we show that the first k solutions generated are indeed the top-k. Referring the example in Fig. 1 with k = 2, that strategy builds solutions S_1 and S_2 in this order.

The rest of the paper is organized as follows. In Section 2, we introduce some preliminary issues. In Section 3 we overview the proposed approach to KS, while in Section 4 we illustrate the approach strategies in more 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 Preliminary Issues

This section states the problem we address and introduces some preliminary notions and terminology. RDF datasets are naturally represented as labeled directed graphs.

Definition 1 (RDF Data Graph). *An RDF data graph is a labeled directed graph G composed by a tuple $G = \{V, E, \Sigma_V, \Sigma_E, L_G\}$ 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.*

Intuitively, the problem of KS over RDF is addressed by exploring the dataset to find sub-graphs holding information relevant to the query. We follow the traditional Information Retrieval approach to value matching adopted in full-text search for semantic query expansion. This involves syntactic and semantic similarities to support an imprecise matching. Since this is not a contribution of our work, we will not discuss it further. We define a path as the sequence of vertices and edges from a source to a sink. The *sources* of a graph are those nodes with no in-going edges and the *sinks* are the nodes with no out-going edges.

Definition 2 (Path). *Given a graph $G = \{V, E, \Sigma_V, \Sigma_E, L_G\}$, a path is a sequence $pt = l_{v_1} - l_{e_1} - l_{v_2} - l_{e_2} - \ldots - l_{e_{n-1}} - l_{v_f}$ where $v_i \in V$, $e_i \in E$, $l_{v_i} = L_G(v_i)$, $l_{e_i} = L_G(e_i)$, and v_1 is a source and v_f is a sink.*

If a source is not present, a fictitious one can be added. For instance, the graph in Fig. 1 has two sources: *pub1* and *pub2*. An example of path is $p_i =$ pub1-author-aut1-name-Bernstein. Obviously, at running time we are interested in the paths relevant to the query, that is, the paths containing at least one vertex matching a keyword of the query. In particular, as assumed in [12], users enter keywords corresponding to attribute values, that are necessarily within the sink's labels. Under this assumption, we do not search URIs: this is not a limitation because nodes labeled by URIs are usually linked to literals, which represent verbose descriptions of such URIs. We index all paths starting from a source and ending with a sink. 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 wildcard #. In the example again using Fig. 1, the template t_{p_2} associated to p_2 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 node, we say that there is an intersection between p_i and p_j and we indicate it with $p_i \leftrightarrow p_j$. Finally, a solution S 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 (Solution). *A solution S is a set of paths $p_1, p_2, \ldots, p_n$ where $\forall p_a, p_b \in S$ there exists a sequence $[p_a, p_{w_1}, \ldots, p_{w_m}, p_b]$, with $m < n$, such that $p_{w_i} \in S$, $p_a \leftrightarrow p_{w_1}$, $p_b \leftrightarrow p_{w_m}$, and $\forall i \in [1, m\text{-}1] : p_{w_i} \leftrightarrow p_{w_{i+1}}$.*

Ranking and monotonicity Given the query Q_1 over the graph illustrated in Fig. 1, the solutions are represented by $S_1 = \{p_1, p_2, p_3\}$ and $S_2 = \{p_4, p_5\}$. Intuitively, S_1 is more relevant than S_2 because it includes more terms from the input query. To assess the relevance of a solution S for a query Q, a scoring function $score(S, Q)$ is adopted. It returns a number that is greater when the solution is more relevant. Then, the ranking is given by ordering the solutions according to their relevance. We say that a ranking is monotonic if the i-th solution is more relevant than the $i + 1$-th solution. Consequently, a query answering process is monotonic if it generates the solutions respecting a monotonic ranking (i.e. the solution of the i-th step is always more relevant than that of the $i + 1$-th step). In the following sections, we will use the notation $score(p, Q)$ and $score(S, Q)$ to evaluate the relevance of a path p and of a solution S with respect to the query Q, respectively. We remark that, unlike all current approaches, we are independent from the scoring function. In fact, we do not impose a monotonic, aggregative nor an "ad-hoc for the case" scoring function. Without giving further details, for the running example and for the experiments we used the scoring function presented in [4].

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 $S_1, S_2, \ldots, S_k$ to Q.

3 Keyword Search over RDF

This section overviews our approach to keyword search over RDF and discusses the conditions under which the solution generation process exhibits a monotonic behavior with respect to the score of the solutions.

Overview. Let G be an RDF data graph and Q a KS query over it. Our approach provides two main phases: the *indexing* (done off-line), in which all the paths of G are indexed, and the *query processing* (done on-the-fly), where the query evaluation takes place. The first task is described in more detail in [2]. 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 in G by exploiting the index and the best solutions are generated from P. An important feature of this phase is the use of the scoring function while computing the solutions. This phase is performed by the following two main tasks:

Clustering. In this task we group the paths of P into clusters cl_i according to their template, and we return the set $\mathcal{CL}$ of all clusters. As an example, given the

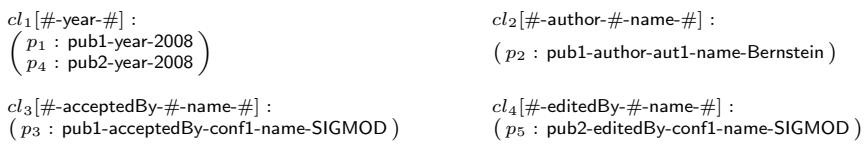


Fig. 2. Clustering of paths.

query Q_1 and the data graph G_1 of Fig. 1, we obtain the clusters shown in Fig. 2. 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 $O(|P|)$: we must only execute $|P|$ insertions into $\mathcal{CL}$ at most.

Building. The last task aims at generating the most relevant solutions by combining the paths in the clusters built in the previous step. This is done by picking and combining the paths with greatest score from each cluster, i.e. the most promising paths. Note that by building solutions with paths from different clusters we diversify the solution content since we do not include homogeneous data, i.e. 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 solution or not. In particular, two different strategies are exploited as follows.

1. *Linear strategy*: guarantees a linear time complexity with respect to the size of the input. Basically, the final solutions are the connected components of the most relevant paths of the clusters.
2. *Monotonic strategy*: generates the solutions in order according to their relevance in a quadratic time complexity with respect to the size of the input. As the linear strategy, it computes the connected components from the most relevant paths in the clusters. Unlike the previous strategy, the path interconnection is not the only criterion to form a solution. At this point every connected components is locally analyzed to check if it fulfills the monotonicity, i.e. we check if the solution we are generating is the optimum. This check is supported by the so called τ-*test*, which is explained in the next Section 3. Furthermore, we derived a variant of this strategy that, reducing a bit the quality of the results, is able to optimize the analysis guaranteeing the execution in linear time w.r.t. the size of the input.

Monotonic Generation. Monotonicity when building the result set represents a significant challenge in keyword search systems. This means returning the optimum solution at each generation step instead of enduring the processing of blocks of candidate solutions and then selecting the optimum. The second strategy relies on the Theorem 1 to guarantee the monotonicity of the building. It requires to verify the two following properties, i.e. Property 1 and Property 2, on the scoring function. Our strategy is independent from such implementation: it works with any scoring function as long as it satisfies the properties below. Furthermore, the two properties are very general and in fact, they are fulfilled

by the most common IR based functions. It is possible to prove that the *pivoted normalization weighting method* (SIM) [11], which inspired most of the IR scoring functions, satisfy Properties 1 and 2. For the sake of simplicity, we discuss the properties by referring to the data structures used in this paper.

Property 1 *Given a query Q and a path p, $score(p, Q) = score(\{p\}, Q)$.*

This property states that the score of a path p is equal to the score of the solution S containing only that same path (i.e. $\{p\}$). It means that every path must be evaluated as the solution containing exactly that path. Consequently we have that, if $score(p_1, Q) > score(p_2, Q)$ then $score(\{p_1\}, Q) > score(\{p_2\}, Q)$. Analogously, extending Property 1 we provide the following.

Property 2 *Given a query Q, a set of paths P in which p_β is the more relevant path (i.e. $\forall p_j \in P$ we have that $score(p_\beta, Q) \geqslant score(p_j, Q)$) and P^* is its power set, we have $score(S = P_i, Q) \leq score(S = \{p_\beta\}, Q)$ $\forall P_i \subseteq P^*$.*

In other words, given the set P containing the candidate paths to be included in the solution, the scores of all possible solutions generated from P (i.e. P^*) are bounded by the score of the most relevant path p_β of P. This property is coherent and generalizes the Threshold Algorithm (TA) [6]. Contrarily to TA, we do not use an aggregative function, nor we assume the aggregation to be monotone. TA introduces a mechanism to optimize the number of steps n to compute the best k objects (where it could be n > k), while our framework produces k optima solutions in k steps. To verify the monotonicity we apply a so-called *τ-test* to determine which paths of a connected component *cc* should be inserted into an optimum solution $optS \subset cc$. The *τ-test* is supported by Theorem 1. Firstly, we have to take into consideration the paths that can be used to form more solutions in the next iterations of the process. In our framework they are still within the set of clusters $\mathcal{CL}$. Then, let us consider the path p_s with the highest score in $\mathcal{CL}$ and the path p_y with the highest score in $cc \setminus optS$. Then we define the threshold τ as $\tau = max\{score(p_s, Q), score(p_y, Q)\}$. The threshold τ can be considered as the upper bound score for the potential solutions to generate in the next iterations of the algorithm. Now, we provide the following:

Theorem 1. *Given a query Q, a scoring function satisfying* Property 1 *and* Property 2*, a connected component cc, a subset $optS \subset cc$ representing an optimum solution and a candidate path $p_x \in cc \setminus optS$, $S = optS \cup \{p_x\}$ is still optimum iff $score(S, Q) \geq \tau$.*

Necessary condition. Let us assume that $S = optS \cup \{p_x\}$ is an optimum solution. We must verify if the score of this solution is still greater than τ. Reminding to the definition of τ, we can have two cases:

- $\tau = score(p_s, Q) > score(p_y, Q)$.
  In this case $score(p_s, Q)$ represents the upper bound for the scoring of the possible solutions to generate in the next steps. Recalling the Property 1, we have $score(p_s, Q) = score(S' = \{p_s\}, Q)$. Referring to

the PROPERTY 2, the possible solutions to generate will present a score less than $score(S' = \{p_s\}, Q)$: $S = optS \cup \{p_x\}$ is optimum. Therefore, $score(S = optS \cup \{p_x\}) \geq \tau$.

- $\tau = score(p_y, Q) > score(p_s, Q)$.
  In a similar way, $score(S = optS \cup \{p_x\}, Q) \geq \tau$.

Sufficient condition. Let us consider $score(S = optS \cup \{p_x\}, Q) \geq \tau$. We must verify if $S = optS \cup \{p_x\}$ is an optimum solution. From the assumption, $score(S = optS \cup \{p_x\}, Q)$ is greater than both $score(p_s, Q)$ and $score(p_y, Q)$. Recalling again the properties of the scoring function, the possible solutions to generate will present a score less than both $score(S' = \{p_s\}, Q)$ and $score(S' = \{p_y\}, Q)$. Therefore, $S = optS \cup \{p_x\}$ is an optimum solution. □

4 Building Strategies

Given the query Q and the set P of paths matching the query Q, we compose those paths to generate the top-k solutions. As said in the previous section, we organize such paths into clusters. In the following we discusses two strategies to compose the paths organized in the set $\mathcal{CL}$ of clusters.

4.1 The Linear Strategy

Given the set of clusters $\mathcal{CL}$, the building of solutions is performed by generating the connected components `cc` from the most promising paths in $\mathcal{CL}$ as shown in Algorithm 1.

Algorithm 1: Building solutions in linear time

```
   Input  : The map CL, a number k.
   Output: A list S of k solutions.
 1 S ← ∅;
 2 while |S| < k and CL is not empty do
 3     first_cl ← ∅;
 4     cc ← ∅;
 5     foreach cl ∈ CL do
 6         first_cl ← first_cl ∪ cl.DequeueTop() ;
 7     cc ← FindCC(first_cl );
 8     s ← ∅;
 9     foreach cc ∈ cc do
10         s.Enqueue(cc);
11     S.InsertAll(s.DequeueTop(k-|S |)) ;
12 return S;
```

The algorithm iterates k times at most to produce the best k solutions (i.e. a list $\mathcal{S}$). At each iteration, we initialize a set `first_cl` with the best paths from each cluster, that is the paths with the highest score (lines [3-6]). `DequeueTop` retrieves the top paths from `cl`, i.e. all paths having the same (top) score. Referring again to the example of Fig. 1, in the first iteration we

have `first_cl` = $\{p_1, p_2, p_3, p_5\}$. Out of `first_cl` we compute the connected components `cc` (line [7]), each of which represents a solution. For the example, $\{p_1, p_2, p_3, p_5\}$ represents a single connected component cc_1. At the second iteration, we have `first_cl` = $\{p_4\}$ and thus, $cc_2 = \{p_4\}$. Then, all generated connected components are included into a priority queue `s`, in order with respect to the score. Finally, through a variant of `DequeueTop`, we insert the top n elements (i.e. n = k -$|\mathcal{S}|$) of `s` into $\mathcal{S}$ (line [11]). The execution concludes when k solutions are produced (i.e $|\mathcal{S}| <$ k) or $\mathcal{CL}$ becomes empty.

Computational Complexity. Algorithm 1 produces the best-k solutions in linear time with respect to the number I of paths matching the input query Q: it is in $O(\text{k}\times I) \in O(I)$. In the worst case, the algorithm iterates k times. The execution in lines [4-5] is $O(|(CL)|) \in O(I)$. Then we have to execute `FindCC` that is $O(I)$ since each path knows which are the other intersecting paths. Finally, both the executions in lines [9-11] and line [12] are in $O(I)$ (i.e. at most we have to make I insertions). Therefore, the entire sequence of operations in Algorithm 1 is in $O(\text{k}\times I) \in O(I)$.

Ranking. Observing the solutions of the running example, S_1 contains the unnecessary p_5, while S_2 is partially incomplete (i.e. it should include p_5). Such strategy tends to produce solutions *exhaustive* but not optimally *specific*, that is to include all relevant information matching the query but not optimally limiting the irrelevant ones. Moreover the solution generated at each step may not be the optimum solution, i.e. the strategy is not monotonic. In fact, it may happen a generation of a sequence of two solutions S_i and S_{i+1} where $score(S_{i+1}, Q) > score(S_i, Q)$. The next section discusses the conditions under which the solution generation process exhibits a monotonic behavior with respect to the score of the solutions.

4.2 The Monotonic Strategy

To generate the top-k solutions guaranteeing monotonicity, differently from Algorithm 1, the building algorithm (Algorithm 2) introduces an exploration procedure to analyze the connected components of the most relevant paths (line [9]).

The function `MonotonicityExploration` (Algorithm 3) finds the best solutions `CCOpt` in `cc` by launching the analysis over each connected component *cc*.

Monotonicity Analysis. Algorithm 4 checks if the solution we are generating is (still) optimum, thus, preserves the monotonicity. It is a recursive function that generates the set `OptSols` of all solutions (candidate to be optimum) by combining the paths in a connected component *cc*. At the end it returns a solution `optS` given by the maximal and optimum subset of paths in *cc*. It takes as input the connected component *cc*, the current optimum solution `optS` and the top path p_s contained in $\mathcal{CL}$.

If *cc* is empty, we return `optS` as it is (lines [1-2]). Otherwise, we analyse all paths $p_x \in cc$ that present an intersection with a path p_i of `optS` ($p_x \leftrightarrow p_i$). If there is not any intersection then `optS` is the final optimum solution (lines [6-7]).

Algorithm 2: Monotonic Building of top-k solutions

```
Input  : A list 𝒞ℒ of clusters, a number k.
Output: A List 𝒮 of k solutions.
1  while |𝒮| < k do
2      first_cl ← ∅;
3      cc ← ∅;
4      foreach cl ∈ 𝒞ℒ do
5          first_cl ← first_cl ∪ cl.DequeueTop() ;
6      cc ← FindCC(first_cl );
7      if 𝒞ℒ is not empty then
8          p_s ← getTopPath(𝒞ℒ );
9          BSols ← MonotonicityExploration(cc, 𝒞ℒ, p_s);
10         𝒮.InsertAll(BSols ) ;
11     else
12         foreach cc ∈ cc do
13             sol ← newSolution(cc);
14             ccSols.Enqueue(sol );
15         𝒮.InsertAll(ccSols.DequeueTop(k-|𝒮 |)) ;
16 return 𝒮;
```

Algorithm 3: Monotonicity Exploration

```
Input  : A set cc of connected components, a list 𝒞ℒ of clusters, a path p_s.
Output: A list of solutions BSols.
1  CCOpt ← ∅;
2  foreach cc ∈ cc do
3      CCOpt.Enqueue( MonotonicityAnalysis(cc, ∅, p_s));
4  BSols.InsertAll(CCOpt.DequeueTop());
5  InsertPathsInClusters(CCOpt, 𝒞ℒ );
6  return BSols ;
```

Otherwise, for each p_x, we calculate τ (line [9]), through the function `getTau`, and then execute the τ-*test* on each new solution `optS'`, that is `optS` $\cup \{p_x\}$. If `optS'` satisfies the τ-*test* (line [11]), then it represents the new optimum solution: we insert it into `OptSols` and we invoke the recursion on `optS'` (line [12]). Otherwise, we keep `optS` as optimum solution and skip p_x (line [14]). At the finish, we want the optimal solution that is not a subset of any other. This is done by selecting the best and maximal solution `optS` from `OptSols` by using `TakeMaximal` (line [15]). Let us consider our running example. As with the linear strategy, in the first iteration of the algorithm we start from `first_cl` $= \{p_1, p_2, p_3, p_5\}$. By using the scoring function in [4], the paths of `first_cl` have scores 2.05, 1.63, 1.6 and 1.49 respectively. Now the exploration considers all possible combinations of these paths to find the optimum solution(s). Therefore, at the beginning we have `optS` $= \{p_1\}$, since p_1 has the highest score, and p_s is p_4. The value of τ is 1.86. The algorithm will then retrieve the following admissible optima solutions: $S'_1 = \{p_1, p_2, p_3\}$, $S'_2 = \{p_1, p_3\}$, and $S'_3 = \{p_1, p_2, p_5\}$. These solutions are admissible because they satisfy the τ-*test* and corresponding paths present pairwise intersections. During computation, the analysis skips solutions $S'_4 = \{p_1, p_2, p_3, p_5\}$ and $S'_5 = \{p_1, p_3, p_5\}$ because they do not satisfy the τ-*test*: the scores of S'_4 and S'_5 are 1.55 and 1.26 respectively, as they are both less than

Algorithm 4: Monotonicity Analysis

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

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

τ. Finally, the function `TakeMaximal` will select S_1' as the final first optimum solution S_1 since it has more paths and the highest score. Following a similar process, at the second round, the algorithm will return $S_2 = \{p_4, p_5\}$ with a lesser score than S_1.

Computational Complexity. Although this analysis achieves our goal, the computational complexity of the result generation process is in $O(I^2)$. As for Algorithm 1, in the worst case the computation iterates k times. In lines [2-6] we follow the same strategy as with Algorithm 1. Therefore, the executions in lines [4-5] and line [6] are in $O(|\mathcal{CL}|) \in O(I)$ and $O(I)$ respectively. Then we have a conditional instruction: if the condition is true, we execute the monotonicity exploration (lines [8-10]), otherwise we consider each connected component $cc \in$ `cc` as a solution to insert into $\mathcal{S}$ (lines [12-15]). As in Algorithm 1, the execution in lines [12-15] is in $O(I)$. In lines [8-10] we call the function `MonotonicityExploration`, that executes the analysis of monotonicity at most I times. This analysis is performed by the recursive function `MonotonicityAnalysis`: in Algorithm 4 the main executions are in lines [9-12] and line [15]. In both the execution is in $O(I)$, since we have I elements to analyze at the most. Since in Algorithm 3 both the operations in line [4] and in line [5] are in $O(I)$, the complexity of the monotonicity exploration is $O(I^2)$. Therefore we conclude that the monotonic strategy is in $O(I^2)$.

Linear Monotonic Strategy. To reduce the complexity of the monotonic strategy, we provide a variant of the monotonicity analysis (i.e. `LinearMonotonicityAnalysis`) that reaches a linear time complexity of the overall process. Without showing the pseudo-code, we can say that this strategy directly selects the best path p_x in cc having an intersection with a path of `optS`. It stops the recursion as soon as the best path does not have in-

tersection with a path in `optS`. In the worst case `optS` is the initial *cc*. `LinearMonotonicityAnalysis` recurses I times and each execution is $O(1)$, therefore the whole strategy is in $O(I)$. Nevertheless, with respect to the building using Algorithm 4, we can generate more specific solutions and (possibly) less exhaustive, since we compose each solution starting from the most relevant path (i.e. we favor keywords that are more closely connected in graph terms).

Correctness, Complexity and Quality of Results. Our discussion is supported by three measures proposed recently [10]: *exhaustivity* ($\mathcal{EX}$), *specificity* ($\mathcal{SP}$) and *overlap*. Exhaustivity measures the relevance of a solution in terms of the number of contained keywords. Specificity measures the precision of a solution in terms of the number of contained keywords with respect to other irrelevant occurring terms. Overlap measures the redundancy of the information content among the solutions. Clearly, the ideal ranking process balances exhaustivity and specificity while reducing overlap. The linear strategy focuses on maximizing the number of keywords in a solution, and consequently the number of paths, privileging $\mathcal{EX}$ to the detriment of $\mathcal{SP}$. On the other hand, the monotonic strategy tries to balance the number of keywords and the number of paths (i.e. maximizing the former and minimizing the latter); therefore $\mathcal{EX}$ and $\mathcal{SP}$ are perfectly balanced. The linear variant of the monotonic strategy is quite similar, but it privileges $\mathcal{SP}$ to the detriment of $\mathcal{EX}$, focusing only on minimizing the number of paths. Finally, all strategies do not generate overlapping solutions since $\forall cl_i, cl_j \in \mathcal{CL}$, with $i \neq j$, we have that $cl_i \cap cl_j = \emptyset$. It means that a path cannot belong to more than one cluster and moreover, we combine paths from different clusters that gather a different kind of information content. As we will demonstrate experimentally in Section 6, state-of-the-art approaches mainly focus on finding the most exhaustive solutions at the cost of a high level of overlapping. In terms of precision and recall, other approaches tend to privilege the recall (finding the best matches with the query) to the detriment of the precision (i.e. introducing a large number of irrelevant matches, that is *noise* in the result set). Demonstrating the correctness of our approach is straightforward. First of all, our algorithm always terminates. Indeed, (i) the clustering groups a finite set of paths (at most all the data graph G), (ii) the building strategies implement recursive functions to traverse finite sets of paths (clusters or connected components). They also employ sets of visited paths to avoid loops on the analysis. Second, our framework returns a match S in G for Q: all paths in S are paths in G that match at least a keyword of Q. Finally, if there exists a match S in G over Q, our framework is able to discover S. In fact, if there exists S for Q over G, then there exists a set of paths in S matching Q. Since our framework indexes all paths in G, we retrieve with Q those paths. If S is also a top-k solution, then we generate it.

5 Related Work

The most prominent work in the area of keyword search concerns the relational databases. Here, answers are usually trees composed of joined tuples, so-called

joined tuples trees (JTTs). They can be classified in *schema-based* or *schema-free* approaches (see [13] for survey). Schema-based approaches (e.g., [8]) implement a middleware layer that makes use of schema information in order to interpret the query and produce a (possible large) number of relational queries. This interpretation is an NP-complete problem [8] and all the SQL statements produced must be executed but some (could) return empty results, leading to inefficiency, which is likely to worsen with the size of the dataset. Schema-free approaches (e.g., [1,9,7]) are more general as they search, on arbitrary graph-shaped data, the *(minimal) Steiner trees*. In all of these approaches a relevant drawback is that finding a (minimal) Steiner tree is known as an NP-Hard problem. Therefore the algorithms rely on (rather) complex sub-routines or heuristics to calculate approximations of Steiner trees. In the best case, such proposals have polynomial complexity in time. The relational approaches are not suitable to work well on RDF data and therefore new approaches have been proposed [12,14,5]. The work in [12] 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 that best represent information need. Candidate queries are computed exploring the top-k sub-graphs matching the keywords. The approach in [14] relies on a RDFS domain knowledge to convert keywords in query-guides that 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 [5] employs a ranking model based on IR and statistical methods.

6 Experimental Results

We implemented our approach in YAANII, a Java system for keyword search over RDF graphs. In our experiments, we used the benchmark provided by Coffman et al. [3] which provides a standardized evaluation using three datasets of different size and complexity. It employs two well-know datasets, IMDB and WIKIPEDIA, and an ideal contrast due to its smaller size, MONDIAL. We used the RDF versions of all three datasets: the *Linked IMDb* and *Wikipedia3*, while for MONDIAL we converted the SQL dump into RDF ourselves. For each dataset, we run the set of 50 queries provided in [3] (see the paper for details and statistics). Experiments were conducted on a dual core 2.66GHz Intel Xeon, running Linux RedHat, with 4 GB of memory, 6 MB cache, and a 2-disk 1Tbyte striped RAID array, and we used Oracle 11g v2 to manage our index, as described in [2].

Performance and Scalability. For query execution evaluation, we compared the different strategies of our system (i.e. linear $\mathcal{L}$, monotonic $\mathcal{M}$ and the variant linear/monotonic $\mathcal{LM}$), with the most related approaches: SEARCHWEBDB ($\mathcal{SWDB}$) [12], EASE ($\mathcal{EASE}$) [9], and the best performing techniques based on graph indexing, i.e. 1000 $\mathcal{BFS}$ and 300 $\mathcal{BFS}$ that are two configurations of BLINKS [7]. For each dataset, we grouped the queries into five sets (i.e. ten queries per set): each set is homogeneous with respect to the complexity of the queries (e.g., number of keywords, number of results and so on). For each set, we

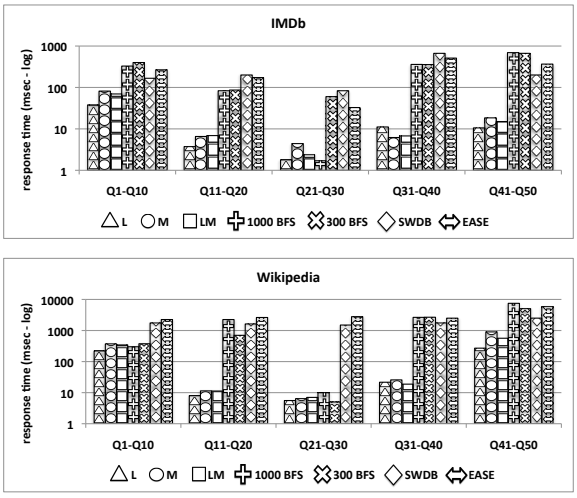
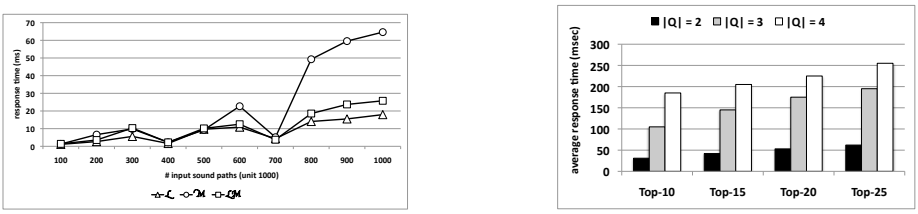


Fig. 3. Response Times on IMDb and Wikipedia.

ran the queries ten times and measured the average response time. The total response time of each query is the time required for computing the top-10 answers. We performed *cold-cache* experiments: we cleared all caches before restarting the various systems and running the queries. The query response times are shown in Fig. 3 (in *ms* and logarithmic scale). Due to space constraints, we report times only on IMDb and Wikipedia, since their much larger size poses more challenges. However the performance on Mondial follows a similar trend. In general EASE and SWDB are comparable with Blinks. Our system performs consistently better (in any strategy) for most of the queries, significantly outperforming the others in some cases (e.g., sets Q21-Q30 or Q31-Q40). This is due to the greatly reduced (time) complexity of the overall process with respect to those that spend a lot of time traversing the graph and computing candidates to be (possible) solutions. An evaluation of the scalability of our system is reported in Fig. 4.(a). In particular, we report the scalability of Yaanii on IMDb. Our system provides a similar behavior on Wikipedia. The figure shows the scalability with respect to the size of the input, that is the number I of paths. Moreover we enriched such experiment by introducing also scalability with respect to the the average size of the query (i.e. $|Q|$), that is the number of keywords, as shown in Fig. 4.(b). In particular we evaluate the impact of the number of keywords to find the top-k (i.e. k $\in \{10, 15, 20, 25\}$) solutions. Also in this case the time grows linearly. The impact of query length is relevant with a higher k.

Effectiveness. We have also evaluated the effectiveness of results. The first measure we used is the reciprocal rank (RR). For a query, RR is the ratio between 1 and the rank at which the first correct answer is returned; or 0 if no correct answer is returned. Fig. 5.(a) shows the mean reciprocal rank of the queries for each system in any dataset. Due to the small size, all systems show comparable performance on the Mondial dataset. Conversely, we have different



(a) (b)

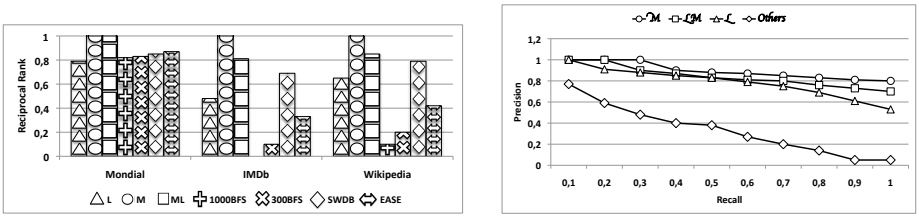
Fig. 4. (a) Scalability w.r.t. #paths on IMDb and (b) w.r.t. the size of Q.

results using IMDb and Wikipedia. As expected, Blinks and Ease performs poorly on this task since they implement a proximity search strategy where the ranking is unable to distinguish solutions containing a single node. SWDB performs well in average because it exploits an IR ranking strategy: usually IR-style search systems prefer larger results supporting the disambiguation of search terms. Our linear strategy $\mathcal{L}$ is comparable with SWDB. This strategy confirms the problems discussed in Section 3: $\mathcal{L}$ favors exhaustive solutions introducing *noise* in the final results (i.e. unnecessary information). On the other hand, the monotonic strategy $\mathcal{M}$ significantly outperforms all others: this strategy is able to return the best result for first (i.e. RR = 1) for all cases. In other words, it demonstrates how much $\mathcal{M}$ balances solutions between being exhaustive and specific. The linear/monotonic strategy $\mathcal{LM}$ shows a similar trend too. We then measured the interpolation between precision and recall to find the top-10 solutions, for each strategy on the queries on all datasets, that is for each standard level r_j of recall (i.e. 0.1, ..., 1.0) we calculate the average max precision of queries in $[r_j, r_{j+1}]$, i.e. $P(r_j) = max_{r_j \leq r \leq r_{j+1}} P(r)$. We repeated this procedure for each strategy. Similarly we calculate the top-10 interpolated precision curve averaged over the systems: Fig. 5.(b) shows the results. As expected, the precision of the other systems decreases dramatically for large values of recall. On the contrary our strategies keeps values within the range [0.6,0.9]. In particular, the monotonic strategy $\mathcal{M}$ presents the highest quality (i.e. a precision in the range [0.8,1]). $\mathcal{LM}$ and $\mathcal{L}$ also present good quality in results.

7 Conclusions and Future Work

In this paper, we presented a novel approach to keyword search query over large RDF datasets, by providing 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. In the worst case, the two strategies present a linear and a quadratic computational cost respectively, whereas other approaches show these results as lower bounds (i.e. best or average cases). Furthermore, we described a variant of the second strategy that reaches both monotonicity and linear complexity. Experimental results confirmed our algorithms and the advantage over other approaches.

This work now opens several directions of further research. From a theoretical point of view, we are investigating algorithms to keyword search over distributed



(a) (b)

Fig. 5. (a) RR measures for all frameworks and (b) Effectiveness of YAANII

environments, retaining the results achieved in this paper. From a practical point of view, we are widening a more synthetic catalogue to index information (e.g., NoSQL technology), optimization techniques to speed-up the index creation and update (mainly DBMS independent) and compression mechanisms.

References

1. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using banks. In: ICDE. pp. 431–440 (2002)
2. 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)
3. Coffman, J., Weaver, A.: An empirical performance evaluation of relational keyword search techniques. TKDE 99(PrePrints), 1 (2012)
4. De Virgilio, R., Cappellari, P., Miscione, M.: Cluster-based exploration for effective keyword search over semantic datasets. In: ER. pp. 205–218 (2009)
5. Elbassuoni, S., Blanco, R.: Keyword search over rdf graphs. In: CIKM. pp. 237–242 (2011)
6. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: PODS. pp. 102–113 (2001)
7. He, H., Wang, H., Yang, J., Yu, P.S.: Blinks: ranked keyword searches on graphs. In: SIGMOD (2007)
8. Hristidis, V., Papakonstantinou, Y.: Discover: Keyword search in relational databases. In: VLDB. pp. 670–681 (2002)
9. Li, G., Ooi, B.C., Feng, J., Wang, J., Zhou, L.: Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In: SIGMOD (2008)
10. Piwowarski, B., Dupret, G.: Evaluation in (xml) information retrieval: expected precision-recall with user modelling (eprum). In: SIGIR. pp. 260–267 (2006)
11. Singhal, A., Buckley, C., Mitra, M.: Pivoted document length normalization. In: SIGIR. pp. 21–29 (1996)
12. 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)
13. Yu, J.X., Qin, L., Chang, L.: Keyword Search in Relational Databases: A Survey. Data(base) Engineering Bulletin 33(1), 67–78 (2010)
14. 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)