ROMA TRE
UNIVERSITÀ DEGLI STUDI

Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79
00146 Roma, Italy

Approximate Querying of RDF Graphs via Path Alignment

Roberto De Virgilio, Antonio Maccioni, Riccardo Torlone

RT-DIA-200-2012 **November 2012**

Università Roma Tre,
Via della Vasca Navale, 79
00146 Roma, Italy

ABSTRACT

A query over RDF data is usually expressed in terms of matching between a graph representing the target and a huge graph representing the source. Unfortunately, graph matching is typically performed in terms of subgraph isomorphism, which makes semantic data querying an hard problem. In this paper we illustrate a novel technique for querying RDF data in which the answers are built by combining paths of the underlying data graph that align with paths specified by the query. The approach is approximate and generates the combinations of the paths that best align with the query. We show that, in this way, the complexity of the overall process is significantly reduced and verify experimentally that our framework exhibits an excellent behavior with respect to other approaches in terms of both efficiency and effectiveness.

1 Introduction

The Linked Data initiative aims at turning the Web into a global knowledge base, where resources are identified by means of URIs, semantically described with RDF, and related through RDF statements. This vision is becoming a reality by the spread of Semantic Web technology, the explosion of social networks, and the availability of more and more linked data sources. However, the rapid increase of semantic data raises in this context severe data management issues [5, 6]. Among them, a major problem lies in the difficulty of users to find the information they need in such huge and heterogeneous repository of semantic data.

In this scenario, approaches to approximate query processing are increasingly capturing the attention of researchers [7, 2, 9, 25, 21] since they relax the matching between queries and data, and thus provide an effective support to non-expert users, who are usually unaware of the way in which data is organized. In particular, since semantic data have a natural representation in the form of a graph, this problem has been often addressed in terms of approximate matching between a small graph representing the query and a very large graph representing the database.

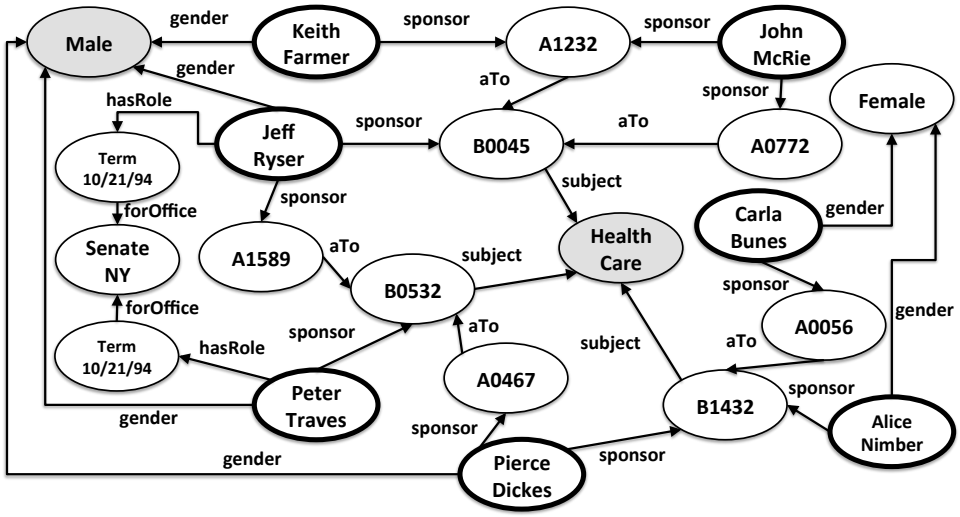
The usual approach to querying a data graph G with a query graph Q is based on searching all the subgraphs of G that are isomorphic to Q. Unfortunately, this problem is known to be NP-complete [11] and the problem is even harder if the matching between query and data is approximate. For this reason, the various approaches to approximate query processing on graph databases rely on heuristics, based on similarity or distance metrics, on the use of specific indexing structures to reduce the complexity of the problem [2, 24, 25], and on fixing some threshold on the maximum number of *hops* (i.e., node/edge additions/deletions needed to perfectly match the query graph with the underlying graph database) that are allowed [9].

In this framework, we propose a novel technique for querying graph-shaped data in an approximate way that combines a strategy for building possible answers with a ranking method for evaluating the relevance of the results as soon as they are computed. The goal is the generation of the best results in the first retrieved answers. We focus in particular on graph queries over large RDF graphs. RDF is the "de-facto" standard language for the representation of semantic information: it encodes Web data as a labeled directed graph in which the nodes represent the resources and links represent semantic relationships between resources. A resource can be an URI, which uniquely identifies an entity in the Semantic Web, or a value (also called literal).

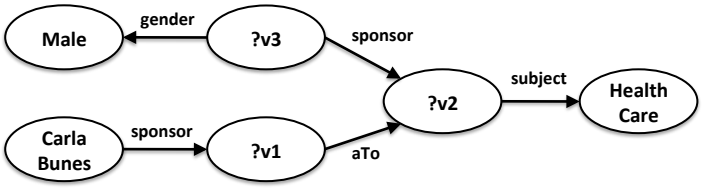
Example 1 *Let us consider the graph G_d depicted in Figure 1, taken from [2]: it represents a simplified portion of the* GovTrack[1] *, a database that stores events occurred in the US Congress. In RDF graphs, nodes represent RDF classes, literals, or URIs, whereas edges represent RDF properties. The graph query Q_1 asks for all amendments ($?v1$) sponsored by* Carla Bunes *to a bill ($?v2$) on the subject of* Health Care *that was originally sponsored by a male person ($?v3$).*

Usually, different paths of the query graph denote different relationships between nodes. For instance, the edges of Q_1 indicate that Male is the gender of someone sponsoring something on the subject Health Care. This simple observation suggests that query

[1] `http://www.govtrack.us`



(a) A RDF data graph G_d



(b) A query Q_1

Figure 1: An example of data and query graph

answering can proceed as follows: first, the query is decomposed into a set of paths that start from a source and end into a sink, then those paths are matched against the data graph, and finally the data paths that best match the query paths are combined to generate the answer.

In our example, this method would decompose Q_1 in the following paths:

$$pq_1: \text{Carla Bunes} \xrightarrow{sponsor} \text{?v1} \xrightarrow{aTo} \text{?v2} \xrightarrow{subject} \text{Health Care}$$
$$pq_2: \text{?v3} \xrightarrow{sponsor} \text{?v2} \xrightarrow{subject} \text{Health Care}$$
$$pq_3: \text{?v3} \xrightarrow{gender} \text{Male}$$

from them, the following paths of G_d would be selected:

$$pd_1: \text{Carla Bunes} \xrightarrow{sponsor} \text{A0056} \xrightarrow{aTo} \text{B1432} \xrightarrow{subject} \text{Health Care}$$
$$pd_2: \text{Pierce Dickens} \xrightarrow{sponsor} \text{B1432} \xrightarrow{subject} \text{Health Care}$$
$$pd_3: \text{Pierce Dickens} \xrightarrow{gender} \text{Male}$$

and those paths, suitably combined, form the answer to the query.

Therefore, we tackle the problem of querying RDF graphs by finding the best combinations of the paths of the data graph that best *align* with the paths of the query graph. Note that in the example above the result is an exact answer to Q_1, but the same strategy can be adopted to generate approximate answers to queries with a suitable relaxation of the notion of alignment between graph paths and data paths. Actually, by using this

technique, the same answer of Q_1 is returned to the query Q_2 reported in Figure 2, for which there is indeed no exact answer.

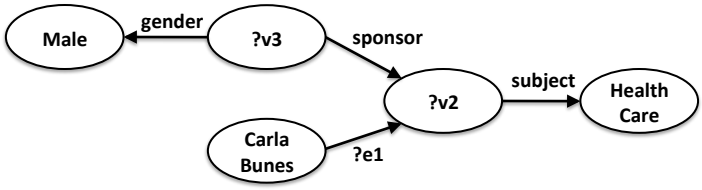


Figure 2: An example of query without exact answers.

The query processing phase first extracts all the paths of data graph G that align with the paths of a query graph Q taking advantage of a special index structure that is built off-line. During the construction, a score function evaluates the solutions in terms of *quality* and *conformity*. The former measures how much the paths retrieved align with the paths in the query. The latter measures how much, in G, the combination of paths retrieved is similar to the combination of the paths in the query. Such strategy exhibits, in the worst case, a quadratic time complexity in the number of nodes of the data graph and our experiments show that the technique scales seamlessly with the size of the input.

In order to test the feasibility of our approach, we have developed a complete system[2] for querying RDF data that implements the above described technique. A number of experiments over widely used benchmarks have shown very good results with respect to other approaches, in terms of both effectiveness and efficiency.

The rest of the paper is organized as follows. In Section 2 we introduce some preliminary notions and definitions. In Sections 3 we illustrate our strategies for graph matching over RDF data and in Section 4 we present the experimental results. In Section 5 we discuss related work and finally, in Section 6, we draw some conclusions and sketch some future works.

2 Preliminary Issues

This section states the problem we address in this paper and introduces some preliminary notions and terminology. We start with the definition of answer to a query in our context and then introduce the data structures and the scoring function that are used in our technique to build and rank the answers to queries.

Problem definition. Let us fix a set Σ_N of node labels and a set Σ_E of edge labels.

Definition 1 (Data Graph) *A data graph $G = \{N, E, L_N, L_E\}$ is a labelled directed graph where N is a set of nodes, $E \subseteq N \times N$ is a set of ordered pairs of nodes, called edges, and L_N and L_E are labeling functions associating an element of Σ_N to each node in N and an element of Σ_E to each edge in E, respectively.*

We focus our attention on (possibly large) RDF databases, which are usually represented as labelled directed graphs in which N and E denote a set of resources and a set

[2] A prototype application is available at `https://www.dropbox.com/sh/d5u1u24qnyqg18f/7Oefq8-qVa`

of properties, respectively. Therefore, given a set $\mathcal{U}$ of *URIs* and a set $\mathcal{L}$ of *literals*, we assume that: $\Sigma_N = \mathcal{U} \cup \mathcal{L}$ and $\Sigma_E = \mathcal{L}$.

Let `VAR` be a set of *variables*, denoted by the prefix "?", a *query graph* Q is defined as follows.

Definition 2 (Query Graph) *A query graph Q is a data graph where $\Sigma_N = \mathcal{U} \cup \mathcal{L} \cup$ `VAR` and $\Sigma_E = \mathcal{L} \cup$ `VAR`.*

A *substitution* for a query graph Q is a function that maps the variables in Q to either URIs or literals. A *transformation* τ on a query graph is a sequence of the following basic update operations: node and edge insertion, node and edge deletion, and labeling modification of both nodes and edges.

Definition 3 (Query answer) *An (approximate) answer to a query graph Q over a data graph G is a subgraph G' of G for which there exists a substitution ϕ and a transformation τ such that $G' = \tau(\phi(Q))$. If τ is empty, G' is an* exact *answer to Q.*

Actually, in our implementation, we have extended this notion by using standard libraries for testing the equality of values based on traditional approaches to full text search (such as stemming). This aspect is however outside the scope of the paper and it will be not discussed further.

Intuitively, an answer $a_1 = \tau_1(\phi_1(Q))$ is more relevant than another an answer $a_2 = \tau_2(\phi_2(Q))$ if τ_1 requires a lower number of operations than τ_2. Moreover, in the context of RDF data in which nodes represent concepts and edges represent relationships, it is useful to associate a weight of relevance to each basic update operation. For instance, it is reasonable that the modification of a label less relevant than a node insertion, since the latter increases the semantic distance between concepts. Therefore, let ω be a function that associates a *weight of relevance* to each basic operation $\odot$. We say that the *cost* γ of a transformation $\tau = \odot_1 \circ \ldots \circ \odot_z$ is $\gamma(\tau) = z \cdot \sum_{i=1}^{z}(\omega(\odot_i))$.

Definition 4 (Relevance of an answer) *An answer $a_1 = \tau_1(\phi_1(Q))$ is more relevant than another answer $a_2 = \tau_2(\phi_2(Q))$ if $\gamma(\tau_1) < \gamma(\tau_2)$.*

Then, given a data graph G and a query graph Q, we aim at finding the top-k answers $a_1, \ldots, a_k$ of Q according to their relevance.

Paths and solutions. We now introduce a number of notions that, in our approach, are used in the construction of the answers to a query. We call *sources* of a graph the nodes with no in-going edges and *sinks* the nodes with no out-going edges. A *full path* in a graph is a path from a source to a sink.

Definition 5 (full path) *Given a data graph $G = \{N, E, L_N, L_E\}$, a full path is a sequence $l_{n_1} - l_{e_1} - l_{n_2} - \ldots - l_{e_{k-1}} - l_{n_k}$ where $l_{n_i} = L_N(n_i)$, $l_{e_i} = L_E(e_i)$, $n_i \in N$, $e_i \in E$, n_1 is a source and n_k is a sink.*

Sources are used as starting points for navigating the graph through full paths. If sources are not present in the given data graph, we promote the *hub* nodes to play this role. A node is an hub in a data graph G if the difference between the number of outgoing edges and the number of the incoming edges is maximum in G.

The data graph in Figure 1 has seven sources (the double-marked nodes) and two sinks (*Health Care* and *Male*, marked in gray). An example of full path is:

$$pd_z = \text{JR-sponsor-A1589-aTo-B0532-subject-HC}$$

where JR and HC denote *Jeff Ryser* and *Health Care*, respectively. The length of a path is the number of nodes occurring in the path, while the position of a node corresponds to its position in the path. For instance, pd_z has length 4 and the node A1589 has position 2. The query Q_1 in Figure 1 has the following full paths:

q_1 : CB-sponsor-?v1-aTo-?v2-subject-HC
q_2 : ?v3-sponsor-?v2-subject-HC
q_3 : ?v3-gender-Male

Our technique tries to generate answers to a query Q by applying substitutions and transformations to full paths of Q. This operation is called alignment.

Definition 6 (Alignment) *Given a data graph G and a query graph Q an alignment is a substitution ϕ and a transformation τ of a full path p of Q such that $\tau(\phi(p))$ is a full path of G.*

We are ready to introduce our notion of solution. We say that a set P of paths of a graph G is a *connected component* of G if, for each pair of paths $p_1, p_2 \in P$, there is a sequence of paths $[p_1, \ldots, p_2]$ in P in which each element has at least a node in common with the following element in the sequence.

Definition 7 (Solution) *Given a query graph Q, a solution of Q over a data graph G is a set of alignments of all the full paths of Q that forms a connected component of G.*

Note that a solution of Q over a data graph G is indeed an answer of Q over G.

Scoring Function. The relevance of solutions is evaluated by means of a scoring function *score* that aims at simulating the relevance of answers (Definition 4). This is motivated by the fact that the latter corresponds to the *graph edit distance* [19] between two graphs, which requires exponential time in the number of nodes of the graph. Conversely *score* can be computed in linear time on the size of data.

The first aspect that this function considers is the quality of alignment between paths of a solution S and paths of a query Q as follows:

$$\Lambda(S, Q) = \sum_{q \in Q} (\lambda(p, q))$$

In this formula, q is a path of Q, p is the path of S that originates from an alignment $\tau \circ \phi$ of q (that is, $p = \tau(\phi(q))$), and λ is a function defined as follows:

$$\lambda(p, q) = (n_N^+ - a \cdot n_N^- - b \cdot n_N^{\curvearrowright}) + (n_E^+ - c \cdot n_E^- - d \cdot n_E^{\curvearrowright}) \quad (1)$$

In this expression: (i) n_N^+ and n_E^+ are, respectively, the number of nodes and edges in common between p and q, (ii) n_N^- and n_E^- are, respectively, the number of nodes and edges of p that are not present in q, and (iii) $n_N^{\curvearrowright}$ and $n_E^{\curvearrowright}$ are, respectively, the number

of nodes and edges inserted in q by τ. Finally, a, b, c and d are parameters that serve to take into account the weights of relevance of the operations in τ (see Definition 4).

The second aspect that the *score* function considers is the conformity between the combination of the paths in the solution and the combination of the paths in the query. This is evaluated as follows:

$$\Psi(S,Q) = \sum_{q_i,q_j \in Q} (\psi(q_i,q_j,p_i,p_j))$$

In this equation, q_i and q_j are paths of Q, p_i and p_j are the paths of S that originate from alignments $\tau_i \circ \phi_i$ and $\tau_j \circ \phi_j$ of q_i and q_j respectively (that is, $p_i = \tau_i(\phi_i(q_i))$ and $p_j = \tau_j(\phi_j(q_j))$), and ψ is a function defined as follows:

$$\psi(q_i,q_j,p_i,p_j) = \frac{|\chi(p_i,p_j)|}{|\chi(q_i,q_j)|} \tag{2}$$

where χ is a function that associates with each pair of paths (p_1,p_2) the set of nodes in common between p_1 and p_2. It follows that $\psi(q_i,q_j,p_i,p_j)$ returns the ratio between the sizes of $\chi(p_i,p_j)$ and $\chi(q_i,q_j)$.

The final score function is then computed as:

$$score(S,Q) = \Lambda(S,Q) + \Psi(S,Q) \tag{3}$$

It turns out that, with a suitable choice of the parameters in Equation 1 that considers the weights of relevance assigned to the basic operations (according to Definition 4), *score* is *coherent* with the notion of relevance of an answer, that is, for each pair of solutions S_1 and S_2 for a query Q such that S_1 is more relevant than S_2 we have that $score(S_1,Q) > score(S_2,Q)$. In fact, let $\odot_N^{\rightarrow}$, $\odot_N^{-}$ and $\odot_N^{\times}$ be basic update operations of node insertion, node deletion, and labeling modification, respectively. Analogously, $\odot_E^{-}, \odot_E^{\rightarrow}$ and $\odot_E^{\times}$ are the respective operations on edges. On these operations, we fix the function ω: (i) $\omega(\odot_N^{-}) = a$, (ii) $\omega(\odot_N^{\rightarrow}) = b$, (iii) $\omega(\odot_E^{-}) = c$ and (iv) $\omega(\odot_E^{\rightarrow}) = d$. We consider, as in other works [15], $\omega(\odot_N^{\times}) = 0$ and $\omega(\odot_E^{\times}) = 0$ because we don't want to penalize the case where the answer gathers more labels than Q.

Now, let us count the number of basic update operations in a transformation τ_i for an answer a_i. In this case we have: n_N^{-} and n_E^{-} are, respectively, the number of nodes and edges of a_i that are inserted in Q, and $n_N^{\rightarrow}$ and $n_E^{\rightarrow}$ are, respectively, the number of nodes and edges updated in Q by τ_i.

The cost of $\gamma(\tau_i)$ is $n_N^{-} \cdot a + n_N^{\rightarrow} \cdot b + n_E^{-} \cdot c + n_E^{\rightarrow} \cdot d$. Let $a_1 = \tau_1(\phi_1(Q))$ and $a_2 = \tau_2(\phi_2(Q))$ be two answers over a query Q. They have two corresponding solutions, S_1 and S_2, still over Q. Considering, from Definition 4, that $a_1 = \odot_1^1 \circ \ldots \circ \odot_z^1$ is more relevant than $a_2 = \odot_1^2 \circ \ldots \circ \odot_y^2$, we have that

$$\gamma(\tau_1) < \gamma(\tau_2) \tag{4}$$

Let us also consider the number n_{N,a_i}^{+} and n_{E,a_i}^{+} of nodes and edges, respectively, in common between an answer a_i and the query Q. $n_{N,a_1}^{+} + n_{E,a_1}^{+} > n_{N,a_2}^{+} + n_{E,a_2}^{+} > 0$ because we have updated or inserted more nodes (edges) in a_2 than in a_1 and then less nodes (edges) can be in common with it. From Equation 4 we derive

$$\gamma(\tau_1) < \gamma(\tau_2)$$

$$\Downarrow$$
$$-\gamma(\tau_1) > -\gamma(\tau_2)$$
$$\Downarrow$$
$$-\gamma(\tau_1) + n^+_{N,a_1} + n^+_{E,a_1} > -\gamma(\tau_2) + n^+_{N,a_2} + n^+_{E,a_2}$$

But $-\gamma(\tau_i) + n^+_{N,a_i} + n^+_{E,a_i} = \Lambda(S_i, Q)$, then

$$\Lambda(S_1, Q) > \Lambda(S_2, Q) \tag{5}$$

that satisfies the hypothesis for the first aspect of *score*. The conformity $\Psi(S_i, Q)$ follows a similar trend than $(n^+_{N,a_i} + n^+_{E,a_i})$. In fact, more are the nodes in common between a solution and a query and more intersections are between the paths in the solution conforming to the intersections between the paths of the query. In our case we have that

$$\Psi(S_1, Q) > \Psi(S_2, Q).$$

Given Equation 1 and Equation 5, it follows that $score(S_1, Q) > score(S_2, Q)$. ▫

3 Path-based Query Processing

3.1 Overview

Let G be a data graph and Q a query graph on it. The approach is composed of two main phases: the *indexing* (done off-line), in which all the full paths of G are indexed, 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 4.

In the second phase, all full paths PD for Q are retrieved in G by exploiting the index and the best solutions are generated from PD by adopting a strategy that guarantees a polynomial time complexity with respect to the size of PD. This task is performed by the following main steps:

Preprocessing. Given a query graph Q, in this step the set PQ of all full paths is computed on the fly by traversing Q from each source to any sinks. We exploit an optimized implementation of the *Breadth-first search* (BFS) traversal. The elements of PQ are organized in the the so-called *intersection query graph* (IG). Each node of IG are full paths of Q while an edge (q_i, q_j) means that q_i and q_j) have nodes in common. For instance, referring to Example 1, PQ consists of the following full paths.

q_1 : Carla Bunes-sponsor-?v1-aTo-?v2-subject-Health Care
q_2 : ?v3-sponsor-?v2-subject-Health Care
q_3 : ?v3-gender-Male

The intersection query graph built from q_1, q_2 and q_3 is depicted in Figure 3. For example, this data structure keeps track of the fact that q_1 and q_2 have nodes in common, i.e. ?v2 and Health Care, as q_2 and q_3 have nodes in common, i.e. only ?v3.

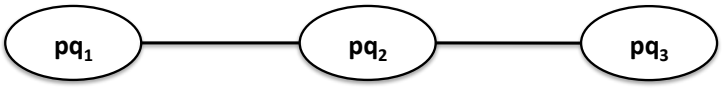


Figure 3: An example of intersection query graph.

Clustering. In the second step we build a cluster for each element q of PQ. Then, we group in the same cluster all the full paths p of G having a sink that matches the sink of q. If q provides a variable in place of the sink, we retrieve the first (constant) value v occurring in q (w.r.t. the end of q, i.e. in the contrary way) and we group in the same cluster all the full paths p of G containing a label matching v. Before the insertion of a full path p in the cluster for q, we evaluate the alignment needed to obtain p from q. This allows us to compute the score of p, i.e. $\lambda(p, q)$. The full paths in a clusters are ordered according to their score with the greater coming first. Note that the same full path p can be inserted in different clusters, possibly with a different score. As an example, given the data graph G_d and the query graph Q_1 of Figure 1, we obtain the clusters shown in Figure 4. In this case clusters cl_1, cl_2 and cl_3 correspond to the full paths q_1, q_2 and q_3 of PQ, respectively; note the scores at the right side of each full path and in particular the full path p_1 occurring in both cl_1 and cl_2 with different scores, i.e. 7 in cl_1 and 5 in cl_2.

$$cl_1 : \begin{pmatrix} p_1: & \text{CB-sponsor-A0056-aTo-B1432-subject-HC [7]} \\ p_2: & \text{JR-sponsor-A1589-aTo-B0532-subject-HC [5]} \\ p_3: & \text{KF-sponsor-A1232-aTo-B0045-subject-HC [5]} \\ p_4: & \text{JM-sponsor-A0772-aTo-B0045-subject-HC [5]} \\ p_5: & \text{JM-sponsor-A1232-aTo-B0045-subject-HC [5]} \\ p_6: & \text{PD-sponsor-A0467-aTo-B0532-subject-HC [5]} \end{pmatrix}$$

$$cl_2 : \begin{pmatrix} p_7: & \text{JR-sponsor-B0045-subject-HC [5]} \\ p_8: & \text{PT-sponsor-B0532-subject-HC [5]} \\ p_9: & \text{AN-sponsor-B1432-subject-HC [5]} \\ p_{10}: & \text{PD-sponsor-B1432-subject-HC [5]} \\ p_{11}: & \text{CB-sponsor-A0056-aTo-B1432-subject-HC [3.5]} \\ p_{12}: & \text{JR-sponsor-A1589-aTo-B0532-subject-HC [3.5]} \\ p_{13}: & \text{KF-sponsor-A1232-aTo-B0045-subject-HC [3.5]} \\ p_{14}: & \text{JM-sponsor-A0772-aTo-B0045-subject-HC [3.5]} \\ p_{15}: & \text{JM-sponsor-A1232-aTo-B0045-subject-HC [3.5]} \\ p_{16}: & \text{PD-sponsor-A0467-aTo-B0532-subject-HC [3.5]} \end{pmatrix}$$

$$cl_3 : \begin{pmatrix} p_{17}: & \text{JR-gender-Male [3]} \\ p_{18}: & \text{KF-gender-Male [3]} \\ p_{19}: & \text{JM-gender-Male [3]} \\ p_{20}: & \text{PD-gender-Male [3]} \end{pmatrix}$$

Figure 4: An example of the clustering step.

Search. The last step aims at generating the most relevant solutions by combining the full paths in the clusters built in the previous step. This is done by picking and combining the full paths with greatest score from each cluster. The intersection query graph allows us to verify efficiently if they form a solution. As an example, given the cluster in Figure 4, the first solution is obtained by combining the full paths

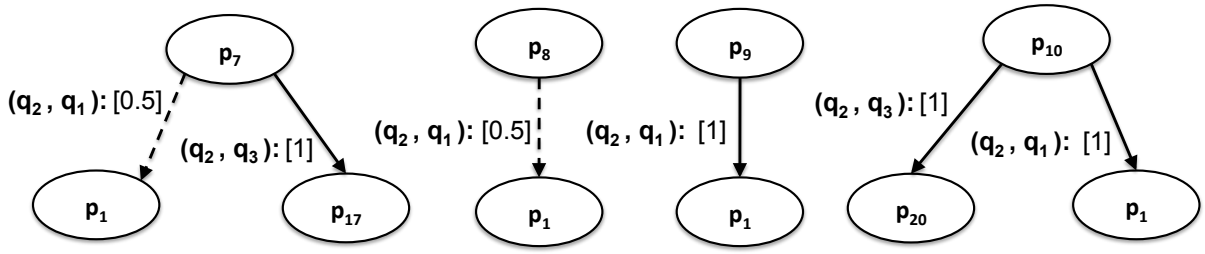


Figure 5: Forest of full paths.

p_1, p_{10} and p_{20} that are the elements with the greatest score in each corresponding cluster and provide the best alignment with the full paths of PQ associated to the clusters.

The most tricky task of the whole process occurs in the third step above. Here, we aim at generating directly the top-k solutions by trying to minimize the number of combinations between full paths. This is done by organizing the combinations of full paths in a forest where nodes represent the retrieved full paths, while edges between full paths means that they have nodes in common. The label of each edge (p_i, p_j) is $\langle (q_i, q_j) : [\psi(q_i, q_j, p_i, p_j)] \rangle$ where q_i and q_j are the full paths corresponding to the clusters where p_i and p_j were included, respectively.

For instance, Figure 5 reports the forest for the full paths with the higher score extracted from the clusters in Figure 4. The label on the edge (p_{10}, p_1) indicates that if p_{10} and p_1 originate from q_2 and q_1, respectively, then $\psi(q_2, q_1, p_{10}, p_1)$ is 1. Conversely, the label on the edge (p_7, p_1) indicates that $\psi(q_2, q_1, p_7, p_1)$ is 0.5. Note that in the forest the edge (p_7, p_1) is dashed since the label is not 1. The tree in the forest with nodes p_1, p_{10} and p_{20} yields the first solution.

The rest of the section describes in more detail the clustering and search steps of the approach.

3.2 Clustering

Given the intersection query graph built in the previous steps (i.e. also the set PQ) and a data graph G, we retrieve and group the full paths from G ending into the sinks of the full paths of PQ, as shown in Algorithm 1.

The set $\mathcal{CL}$ of clusters is implemented as a map where the key is a full path q from PQ and the value is a cluster with all the full paths p ending in the sink of q. Each cluster is implemented as a priority queue of full paths, where the priority is based on the score (descending) associated to each path. In this paper, we use an advanced implementation of priority queues to support costant time complexity for insertion/deletion operations. For each $q \in$ PQ [line 1] we extract the sink sk of q and we retrieve all p from G ending in sk. Such task is performed by the function getPaths [line 3]. Once obtained the set PD, we evaluate the score of each $p \in$ PD with respect to q and we insert p in the cluster cl [lines 4-5]. Finally we insert cl in $\mathcal{CL}$ [line 8].

Different scores could be obtained for the same pair (p, q) if we use different transformations τ (and/or substitutions ϕ); however in this work we consider only RDF graphs. Therefore we employ a specific strategy to apply transformation and substitution: we

Algorithm 1: Clustering of paths

Input : The query full paths PQ, the data graph G.
Output: The set of clusters $\mathcal{CL}$.

```
1 foreach q ∈ PQ do
2     cl ← ∅ ;
3     PD ← getPaths(G, q) ;
4     foreach p ∈ PD do
5         cl.enqueue(p, λ(p, q))
6     CL.put(q, cl) ;
7 return CL ;
```

start from the final node of q and we try to align constant values of q with constant values of p, first of all. Then we insert/delete/modify nodes to compute the final alignment between p and q (i.e. we proceed contrary to the direction of the edges). This operation is linear in the number of nodes of p and q since we compute each node just once. For instance let us consider q_1 and q_2 from the query graph Q_1 and the full path

$$p = \text{CB-sponsor-A0056-aTo-B1432-subject-HC}$$

from G_d. We evaluate the score of p with respect to both q_1 and q_2 as follows

$$\begin{aligned} q_1 &: \text{CB-sponsor-?v1-aTo-?v2-subjec-HC} \\ \tau_1(\phi_1(q_1)) &: \underbrace{\textbf{CB}}\text{-sponsor-}\underbrace{\textbf{A0056}}\text{-aTo-}\underbrace{\textbf{B1432}}\text{-subject-HC} \\ q_2 &: \text{?v3-sponsor-?v2-subject-HC} \\ \tau_2(\phi_2(q_2)) &: \underbrace{\textbf{CB}}\text{-sponsor-}\underbrace{\textbf{A0056}}\text{-}\underbrace{\textbf{aTo-B1432}}\text{-subject-HC} \end{aligned}$$

In this case q_1 requires only a substitution ϕ on the variables while q_2 employs a transformation τ to insert **aTo-B1432** and a substitution ϕ on the variables. In the former case we have $\lambda(p, q_1) = (4 - 0 - 0) + (3 - 0 - 0) = 7$, since $n_V^+ = 4$, $n_E^+ = 3$ and $n_V^- = n_V^{\curvearrowright} = n_E^- = n_E^{\curvearrowright} = 0$. In the latter case $\lambda(p, q_2) = (3 - 0 - b) + (2 - 0 - d)$, since $n_V^+ = 3$, $n_E^+ = 2$, $n_V^- = n_E^- = 0$, and $n_V^{\curvearrowright} = n_E^{\curvearrowright} = 1$. If we set $b = 0.5$ and $d = 1$, we have $\lambda(p, q_2) = 3.5$ (i.e. p has the best alignment with q_1). In the same way, given

$$p' = \text{JR-sponsor-A1589-aTo-B0532-subject-HC}$$

we can calculate $\lambda(p', q_1) = (3 - a - 0) + (3 - 0 - 0)$, since $n_V^- = 1$ due to the mismatch between CB and JR. If we set $a = 1$, $\lambda(p', q_1) = 5$ (i.e. q_1 has a better alignment with p than p').

It is straightforward to demonstrate that the time complexity of the clustering is $|Q| \times O(I)$, where $|Q| = |\text{PQ}|$: we have to execute I insertions into $\mathcal{CL}$ for $|Q|$ times at most.

3.3 Search

Given the set of clusters $\mathcal{CL}$, we retrieve the top-k solutions by generating the connected components from the most promising paths in $\mathcal{CL}$. Algorithm 2 summarizes the entire process of search to retrieve the top-k solutions.

Algorithm 2: Search

Input : number k, clusters $\mathcal{CL}$, intersection query graph IG.
Output: The set of solutions $\mathcal{S}$.

```
1  while (CL ≠ ∅) ∧ (|S| < k) do
2      F ← build( CL, IG );
3      roots ← F.keys;
4      while (|S| < k) ∧ (roots ≠ ∅) do
5          V ← ∅;
6          r ← getMax(roots );
7          roots − {r};
8          s ← Visit(r, getQPath(CL, r ), V, F.get(r ));
9          S.add(s );
10 return S;
```

Until we did not generate k solutions or the set of clusters is not empty [line 1], we build a forest $\mathcal{F}$ [line 2] from the most promising full paths in $\mathcal{CL}$ and provide the top-k solutions by visiting $\mathcal{F}$ [lines 4-8]. As said in Section 3.1, nodes of $\mathcal{F}$ are full paths in $\mathcal{CL}$, while edges (p_i, p_j) means that p_i and p_j have nodes in common. Trees of $\mathcal{F}$ are solutions to the query. In the following we describe in more detail the building and the visiting of $\mathcal{F}$.

Building. Given the set of clusters $\mathcal{CL}$ and the intersection query graph IG, first of all we have a *building* phase generating a *forest* of full paths, as shown in Algorithm 3.

Algorithm 3: Build

Input : clusters $\mathcal{CL}$, intersection query graph IG.
Output: a forest $\mathcal{F}$.

```
1  q ← maxCardinality(IG);
2  cl ← CL.get(q);
3  PD ← cl.dequeueTop();
4  V ← {q};
5  F ← ∅;
6  foreach p ∈ PD do
7      T ← ∅;
8      T.N ← T.N ∪ {p};
9      treeBuild(p, T, CL, IG, q, V);
10     F.put(p,T);
11 return F;
```

Navigating IG we can evaluate the conformity of the solutions while we build them. Algorithm 3 implements a BFS traversal. Therefore we start from the node q of IG with the maximum cardinality [line 1]. For instance, referring to Q_1 of Example 1, the algorithm selects q_2 as starting node. Then we select the cluster cl corresponding to q [line 2] and we dequeue the top full paths PD from cl [line 3]. This task is supported by the function dequeueTop. The path q is inserted in the set V of visited query paths [line

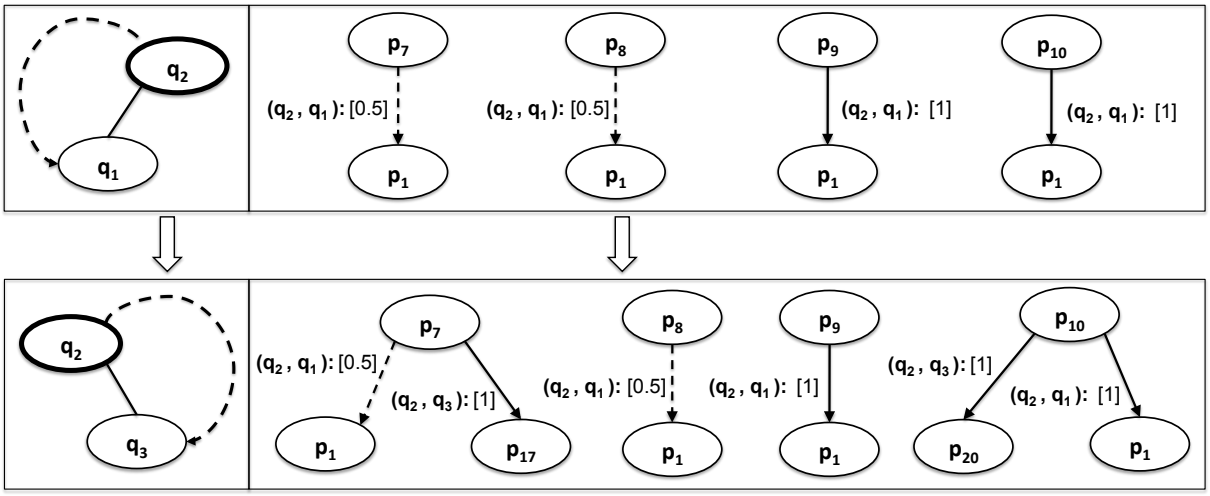


Figure 6: Building of the forest

4]. Referring to our example the cluster cl_2 corresponds to q_2. In this case the top full paths to dequeue are p_7, p_8, p_9 and p_{10}. The full paths PD extracted from the cluster represent the *roots* of the forest $\mathcal{F}$. We implement $\mathcal{F}$ as a map where the keys are full paths, i.e. the roots, and the values are the trees of $\mathcal{F}$. Each tree T of $\mathcal{F}$ is modeled as a graph $\langle N, E\rangle$ where the nodes in N are full paths and each edge $l(n_i, n_j) \in E$ is described in terms of $\langle n_i, n_j, l\rangle$ where l is the label of the edge. For each $p \in$ PD, we build a tree rooted in p by using the procedure treeBuild [line 9]. The procedure is described in details in Algorithm 4.

Algorithm 4: treeBuild

Input : root r, tree T, clusters $\mathcal{CL}$, query full path q, intersection query graph IG,visited V.

```
1 foreach (q, q') ∈ IG.E : q' ∉ V do
2     cl ← CL.get(q');
3     PD ← cl.dequeueTop();
4     foreach p ∈ PD : p ↔ r do
5         T.V ← T.V ∪ {p};
6         e ← ⟨r, p, (q, q') : [ψ(q, q', r, p)]⟩;
7         T.E ← T.E ∪ {e};
8         V ← V ∪ {q'};
9         treeBuild(p, T, CL, IG, q', V);
```

The procedure treeBuild starts to navigate IG from the input query path q. For each edge (q, q'), where q' is not yet visited, we dequeue the top full paths PD from the cluster cl corresponding to q' [lines 2-3]. Then for each full path $p \in$ PD having nodes in common with r (denoted by $p \leftrightarrow$ r), we build the edge (r, p) and the corresponding label $\langle r, p, (q, q') : [\psi(q, q', r, p)]\rangle$, [line 6], and we insert it in the set E of T [line 7]. Finally we include q' in the set V of visited query paths [line 8] and we recursively call the procedure [line 9].

For instance Figure 6 illustrates the building of the forest $\mathcal{F}$ with respect to Example 1. Starting from p_7, p_8, p_9 and p_{10} (i.e. the roots) extracted by the cluster cl_2 (i.e. q_2), we traverse IG: we consider q_1 and q_3. Traversing q_1 we dequeue the top full paths from the cluster cl_1: in this case we have only p_1; p_1 is the successor of all roots since it has nodes in common with them. Therefore, we add an edge between each root and p_1 in $\mathcal{F}$. However each edge presents a different conformity: p_9 and p_{10} provide two nodes in common with p_1 conforming exactly with the query graph Q_1 (i.e. 1); while p_7 and p_8 have one node in common p_1 conforming partially with Q_1 (i.e. 0.5). Similarly, in the traversal of q_3 we dequeue p_{17}, p_{18}, p_{19} and p_{20} from cl_3. In this case only p_{17} and p_{20} have nodes in common with p_7 and p_{10}, respectively, with conformity 1. Once added the last edges to $\mathcal{F}$, the procedure terminates since it visited all the nodes of IG.

The building step (Algorithm 3) is the more involved of the entire process. The number of iterations required by this algorithm requires I iterations, where I is the number of full paths PD retrieved in line 4 that, in the worst case, is proportional to the size of data. In each step of this iteration, there is a call of the procedure treeBuild. Algorithm 4 explores the intersection query graph IG: if h is the depth of IG, in the worst case it processes h times each full path in PD. Then, the complexity of the Algorithm 3 is $O(h \times I^2)$.

Visiting. The last step is to visit the forest $\mathcal{F}$. Algorithm 2 starts the visit from the rootwith the maximum score. The visit implements a *Depth-first search* (DFS) traversal as shown in details in Algorithm 5.

Algorithm 5: Visit

Input : full path p, full path q, visited V, tree T.
Output: set of full paths s.

```
1 if ∄⟨p, p′, l⟩ ∈ T.E then
2     return {p};
3 else
4     foreach (q, q′) ∈ IG.E : q′ ∉ V do
5         v ← getMaxConformity(p,q′,T );
6         T.E − {⟨p, p′, (q, q′) : [v]⟩};
7         V ← V ∪ {q′};
8         return {p′} ∪ Visit(p′,q′,V,T );
```

As in the building step, we exploit the intersection query graph to explore $\mathcal{F}$. Starting from a root p of $\mathcal{F}$ to which the full path q is associated (i.e. p was in the cluster corresponding to q), for each (q, q') in IG we select the successor p' of p to which q' is associated. In particular since we can have multiple p' to which q' is associated, we select the most conforming pair (p, p'). To this aim, the procedure getMaxConformity compute the maximum conformity v associated to a pair (p, p') [line 5]. Once selected p' we include it in the solution s and we recursively call the visit [line 8]. If p has no successors, then we return p.

Referring to Figure 6, we start from roots $= \{p_{10}, p_7, p_9, p_8\}$ (i.e. in order of priority). Then the first solution S_1 dequeues p_{10}. From p_{10}, to S_1 we add p_{20} and p_1, that are the most important full paths to which q_3 and q_1 are associated. Finally we have $S_1 = \{p_{10}, p_1, p_{20}\}$. Similarly we produce in order $S_2 = \{p_7, p_1, p_{17}\}$, $S_3 = \{p_9, p_1\}$ and $S_4 =$

$\{p_8, p_1\}$.

As for the building, the visiting explores IG. Therefore the complexity of this task is $O(k \times h \times I)$, since k times we call the visiting of a tree T, that is $O(h \times I)$.

Table 1: Overall Complexity

clustering	building	visiting	overall
$\|Q\| \times O(I)$	$O(h \times I^2)$	$O(k \times h \times I)$	$O(h \times I^2)$

Overall Complexity. Table 1 summarizes all discussed results. As said above, the building step is the core procedure of the overall process. The complexity of the building determines the overall complexity that is $O(h \times I^2)$, where I is the number of full paths retrieved by the index and h is the depth of the query graph Q. In Section 4, experiments will confirm this study.

4 Experimental Results

We implemented our approach in SAMA[3] , a Java system with a Web front end. We have compared SAMA with three representatives graph matching systems: SAPPER [25], BOUNDED [9] and DOGMA [2]. Experiments were conducted on a dual core 2.66GHz Intel Xeon, running Linux RedHat, with 4 GB of memory and a 2-disk 1Tbyte striped RAID array.

Indexing. To build solutions efficiently, we index the following information: vertices' and edges' labels of the data graph G (for element-to-element mapping) and the full paths ending into sinks, since they bring information that might match the query. The first information enables to locate vertices and edges matching the labels of the query graph, the second allows us to skip the expensive graph traversal at runtime. The indexing process is composed of three steps: (i) hashing of all vertices' and edges' labels, (ii) identification of sources and sinks, and (iii) computation of the full paths. The first and the second step are relatively easy. The third step requires to traverse the graph starting from the sources and following the routes to the sinks. We have implemented an optimized version of the Breadth-First-Search (BFS) paradigm, where independently concurrent traversals are started from each source. Similarly to [2], and differently from the majority of related works (e.g [9]), we assume that the graph cannot fit in memory and that can only be stored on disk. Specifically, we store the index in a GraphDB, that is HyperGraphDB[4] (HGDB) v. 1.1: it models data in terms of hypergraphs.Let us recall an hypergraph H is a generalization of a graph, where an edge can connect any number of vertices. Formally $H = (X, E)$ where X is a set of nodes or vertices, and E is a set of non-empty subsets of X, called *hyperedges*. In other words, E is a subset of the power set of X. This representation allows us to define indexes on both vertices and hyperedges: $X = \{x_m | m \in M\}$ and $E = \{e_f | f \in F, e_f \subseteq X\}$, where each vertex x_m and edge e_f are

[3] A prototype application is available at `https://www.dropbox.com/sh/d5u1u24qnyqg18f/7Oefq8-qVa`

[4] `http://www.kobrix.com/hgdb.jsp`

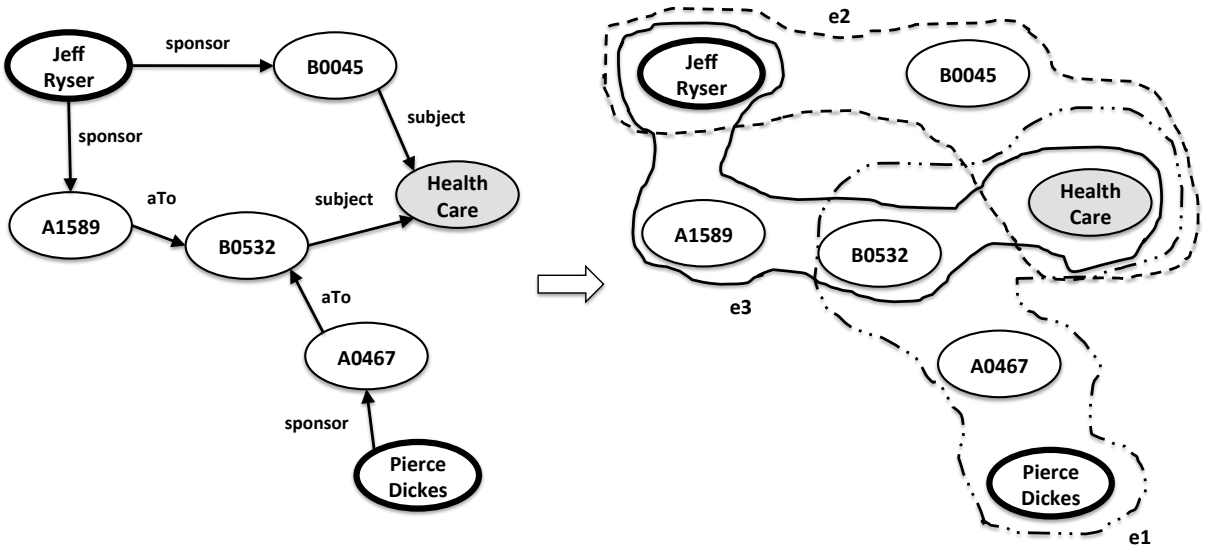


Figure 7: An example to represent a data graph G (left side) in a hypergraph H (right side)

Table 2: HyperGraphDB indexing

DG	#Triples	—HV—	—HE—	t	Space
PBlog	50K	1,5K	96K	1 sec	56 MB
GOV	1M	280K	330K	4 min	340 MB
KEGG	1M	300K	606K	7 min	700 MB
Berlin	1M	320K	700K	10 min	910 MB
IMDB	6M	900K	3M	47 min	1,2 GB
LUBM	12M	1M	15M	102 min	12,9 GB
UOBM	12M	1M	15M	102 min	12,9 GB
DBLP	26M	4M	17M	441 min	23,6 GB

indexed by an index $m \in M$ and $f \in F$, respectively. Figure 7 shows an example of reference.

The matching is supported by standard IR engines (c.f. Lucene Domain index (LDi)[5]) embedded into HGDB. In particular we define a LDi index on the labels of nodes and edges. In this way, given a label, HGDB retrieves all full paths containing data elements matching the label in a very efficient way (i.e. exploiting the cursors). Further, semantically similar entries such as synonyms, hyponyms and hypernyms are extracted from WordNet [10], supported by LDi.

In our experiments we consider real RDF datasets, such as PBlog[6] , GovTrack, KEGG, IMDB [13], DBLP, and synthetic datasets, such as Berlin [1], LUBM [12] and UOBM [16]. Table 2 provides importing information for any dataset: number of triples, number of nodes ($|HV|$) and number of generated hyperedges ($|HE|$) in HGDB, time to create the index on HGDB (t) and memory consumption on disk. In our case, building the index takes hours for large RDF data graphs, due to the demanding traversal on the complete large graph, and requires GB of memory resources on disk to store data

[5] `http://lucene.apache.org/`
[6] `http://www-personal.umich.edu/~mejn/netdata/`

and metadata. However our framework benefits the high performance to retrieve data elements on HGDB, as shown in Table 3. The table illustrates the average response times to retrieve a full path p and all data elements (nodes and edges) associated to p given a label. We performed *cold-cache* experiments (i.e. by dropping all file-system caches before restarting the various systems and running the queries) and *warm-cache* experiments (i.e. without dropping the caches). In particular we refer to the most huge datasets.

Table 3: Average Time to retrieve a full path

DG	Cold (msec)	Warm (msec)
IMDB	0,01	0,005
LUBM	0,04	0,008
DBLP	0,06	0,009

On top of this index organization, to avoid to recompile the entire index on HGDB, we implemented also several procedures to support the maintenance: insertion, deletion and update of new vertices or edges in the data graph G.

Table 4: Index Maintenance Performance

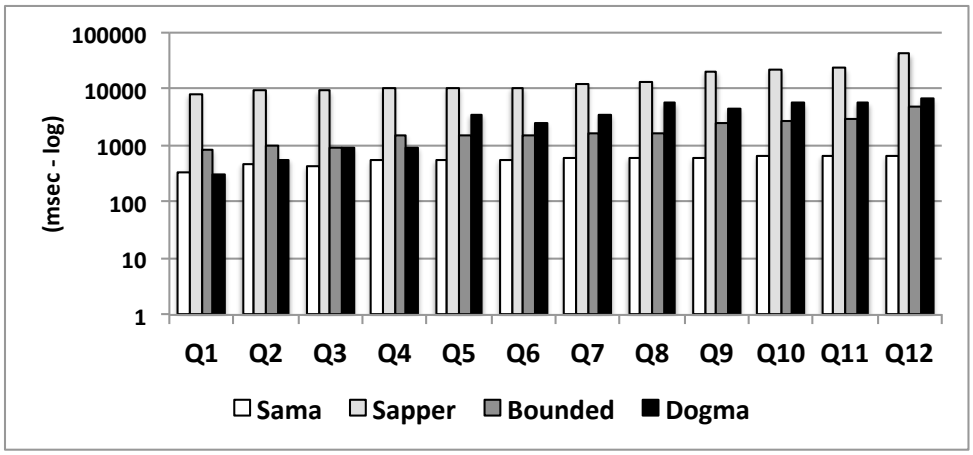
	Insertion	Deletion	Update
Vertex	4.6 (ms)	4.3 (ms)	5.7 (ms)
Edge	11.4 (ms)	22.1 (ms)	6.3 (ms)

Update operations perform well, as demonstrated in Table 4. We inserted and updated 100 vertices (edges) and then we deleted them. We measured the average response time (ms) to insert/delete/update one vertex (edge). Once the index is built the first time, the index can be maintained easily as the maintenance operations satisfy practical scenarios of frequent update of the dataset.

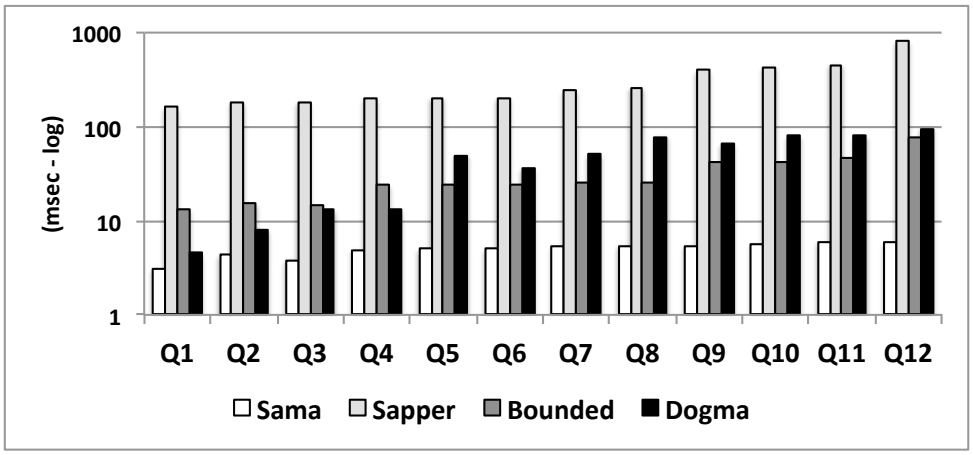
Query Execution. In this experiment, for each indexed dataset we formulated 12 queries in SPARQL of different complexities (i.e. number of nodes, edges and variables). We ran the queries ten times and we measured the average response time, in ms and logarithmic scale. Precisely, the total time of each query is the time for computing the top-10 answers, including any *preprocessing*, *execution* and *traversal*. We performed both cold-cache and warm-cache experiments. To make a comparison with the other systems, we reformulated the 12 queries by using the input format of each competitor. In SAMA we set the coefficients of the scoring function as follows: $a = 1$, $b = 0.5$, $c = 2$ and $d = 1$. Due to space constraints, we cannot describe in detail results on every dataset, neither the query lists[7] . Therefore we show the behavior of all systems with respect to LUBM, the most representative in terms of number of triples and complexity. The query run-times are shown in Figure 8.

In general BOUNDED performs better than DOGMA, while SAPPER is less efficient. SAMA performs very well with respect to all competitors: it is supported by the index that retrieves all needed data elements in an efficient way (i.e. skipping data graph traversal at runtime and supporting parallel implementations). In particular Figure 9 shows

[7] At `https://www.dropbox.com/sh/d5u1u24qnyqg18f/70efq8-qVa` you can find the complete set of queries



(a) cold-cache



(b) warm-cache

Figure 8: Average response time on LUBM: bars refer each system, using different gray scales, i.e. from SAMA, white bars, to DOGMA, black bars

the cumulative time percentage of each step (i.e. preprocessing, clustering, building and visiting) in our framework. In any query, the most amount of time is spent for the preprocessing step (i.e. 89% of the cumulative amount of time in average): we have to traverse the query graph Q and extract all full paths, and to retrieve all sound paths. Of course a decentralized environment could relevantly limit this time consumption. However time consumption percentages of the main steps shows the efficiency of our search algorithm and demonstrate the feasibility of the approach: we compute simple alignments between paths. The other datasets provide a very similar behavior.

Another aspect we test is the scalability of our approach. In particular this experiment demonstrates the overall polynomial time complexity of SAMA, i.e quadratic time. Figure 10 shows the flexibility of SAMA with respect to both the number I of full paths and the number $|Q|$ of full paths in the query Q. For each couple $(I, |Q|)$ we depict the average response time (msec) referring to cold-cache experiments: the number is enclosed in a circle and are scaled proportionally to the number (i.e. warm-cache experiments follow the same trend). The size of each circle is perfectly linear with the growth of both I and $|Q|$: this tells us that our approach is almost linear with respect to both the measures. Such aspect is analyzed in more detail by evaluating the scalability of SAMA with respect

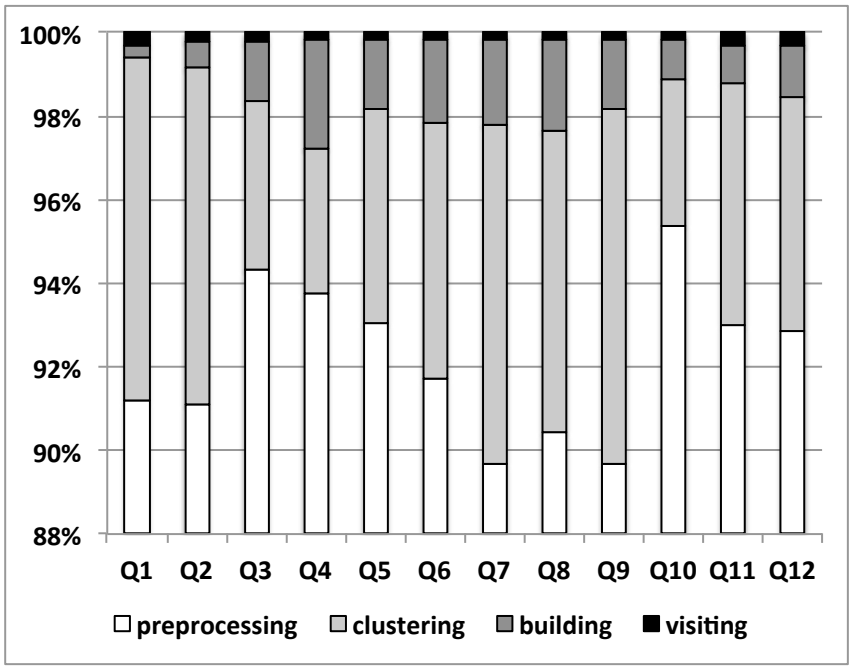


Figure 9: Cumulative Time Percentage of each step

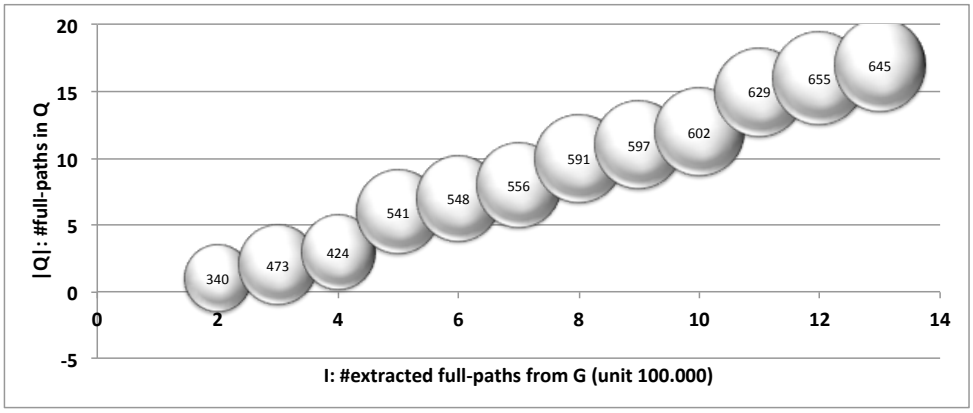


Figure 10: Flexibility of Sama on LUBM

to I, as illustrated in Figure 11 (i.e. it refers to cold-cache experiments). The diagram displays the trend-line and the interpolated equation: the behavior of Sama is polynomial with respect to the time complexity; in particular the trend-line always approximate a quadratic trend (i.e. we have the same for warm-cache experiments).

Effectiveness. The last experiment evaluates the effectiveness of Sama and of the other competitors. 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. In any dataset, for all 12 queries we obtained RR=1. In this case the monotonicity is never violated. To make a comparison with the other systems we inspected the matches found in terms of the solutions returned. Figure 12 shows the effectiveness of all systems on LUBM, where we run the queries without imposing the number k of solutions.

In this case Sama and Sapper always identify more meaningful matches than both

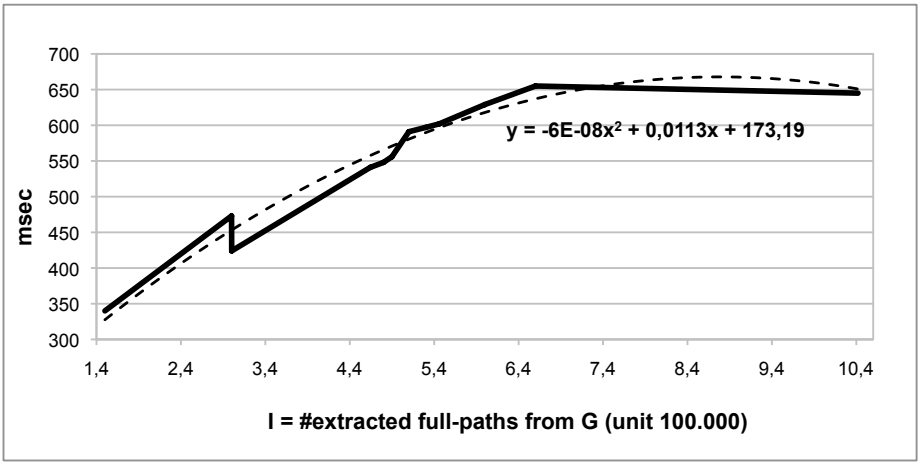


Figure 11: Scalability of SAMA on LUBM with respect to number I of extracted full paths from G

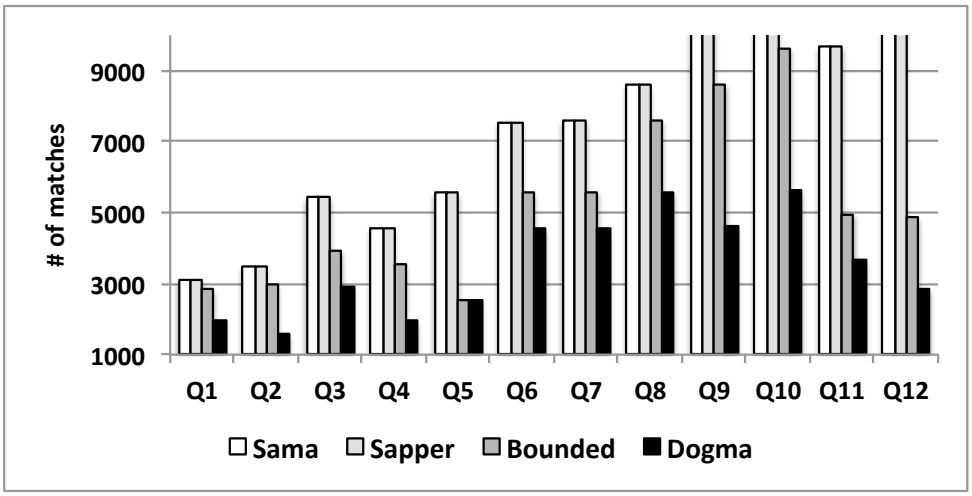


Figure 12: Effectiveness on LUBM: bars refer each system, using different gray scales, i.e. from SAMA, white bars, to DOGMA, black bars

BOUNDED and DOGMA. This is due to the approximation operated by SAMA and SAPPER with respect to the others. We remind that the evaluation of the matches was performed by experts of the domain (e.g. LUBM). Finally, to support the meaningful of results, we measured the interpolation between precision and recall. Figure 13 shows the results on LUBM: for SAMA we depict three different trends with respect to the range of $|Q|$. As to be expected, queries with limited number of full paths presents the highest quality (i.e. a precision in the range [0.5,0.8]). More complex queries decrease the quality of results, due to more data elements retrieved by the approximation, presenting good quality though. Such result confirms the feasibility of our system. The effectiveness on the other datasets follows a similar trend. On the other hand, as to be expected, the precision of the other systems dramatically decreases for large values of recall: BOUNDED and DOGMA do not exploit an imprecise matching, while SAPPER introduces *noise* (i.e. not interesting approximate results) in high values of recall.

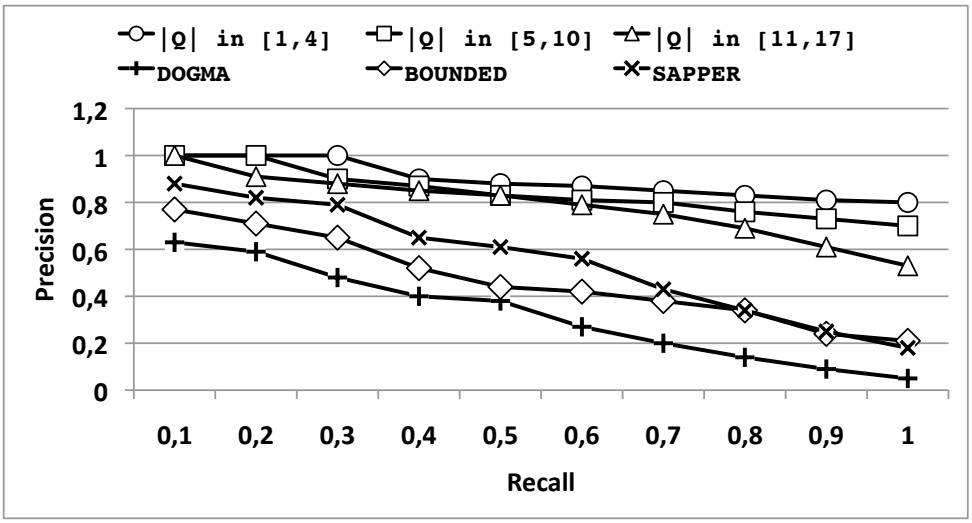


Figure 13: Effectiveness on LUBM: Precision and Recall of Sama

5 Related Work

Many efforts, from different research fields, have focused on graph matching [11]. A first category of works relies on subgraph isomorphism [26]. However the well-known intractability of the problem inspired approximate approaches to simplify the problem [11]. A second category of works focuses on the adoption of special indexes. In particular, several approaches have proposed in-memory structures for indexing the nodes of the data graph [20], while others have proposed specific indexes for the efficient execution of SPARQL queries and joins [17]. In addition, other proposals tackle the problem by indexing graph substructures (e.g. paths, frequent subgraphs, trees). Typically, these indexes are exploited in problems dealing with graph matching, to filter out graphs that do not match the input query. Approaches in this area can be classified in graph indexing and subgraph indexing. In graph indexing approaches, such as gIndex [22], TreePi [23], and FG-Index [4], the graph database consists of a set of small graphs. The indexing aims at finding all the database graphs that contain or are contained in a given query graph. On the other hand, subgraph indexing approaches, such as DOGMA [2] and SAPPER [25], aim at indexing large database graph, with the goal of finding efficiently all (or a subset of) the subgraphs that match a given query. Finally there are works on reachability [14, 18] and distance queries [3] based on testing the existence of a path between two nodes in the graph and on the evaluation of the distance between them. In this context, techniques based on subgraph isomorphism [8] have been proposed but since the problem is `NP-complete`, graph simulation techniques has been used to make graph matching tractable. An interesting approach is proposed in [9] where the authors reformulate the query graph in terms of a bounded query in which an edge denotes the connectivity of nodes within a predefined number of hops. This guarantees a cubic time complexity for the graph matching problem.

All these approaches differ quite a lot from our method. Indeed, we tackle the problem using a technique that takes into account the structural constraints on how different relations between nodes, i.e. full paths, have to be correlated. Moreover, the matching between the data graph and the query graph relies on the tractable problem of alignment between paths.

6 Conclusion and Future Work

In this paper we have presented a novel approach to approximate querying of large RDF data sets. The approach is based on a strategy for building the answers of a query by selecting and combining paths of the underlying data graph that best align with paths in the query. A ranking function is used in query answering for evaluating the relevance of the results as soon as they are computed. In the worst case our technique exhibits a quadratic computational cost with respect to the size of the input and experimental results show that it behaves very well with respect to approaches in terms of both efficiency and effectiveness. This work opens several directions of further research. From a conceptual point of view, we intend to introduce improvements on the construction of solutions and on the on-line computation of the scoring function. From a practical point of view, we intend to implement the approach in a Grid environment (for instance using Hadoop/Hbase) and develop optimization techniques to speed-up the creation and the update of the index, as well as compression mechanisms for reducing the overhead required by its construction and maintenance.

References

[1] Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[2] Matthias Bröcheler, Andrea Pugliese, and V. S. Subrahmanian. Dogma: A disk-oriented graph matching algorithm for rdf databases. In *ISWC*, pages 97–113, 2009.

[3] Edward P. F. Chan and Heechul Lim. Optimization and evaluation of shortest path queries. *VLDB J.*, 16(3):343–369, 2007.

[4] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872, 2007.

[5] R. De Virgilio, F. Giunchiglia, and L. Tanca, editors. *Semantic Web Information Management - A Model-Based Perspective*. Springer Verlag, 2010.

[6] R. De Virgilio, F. Guerra, and Y. Velegrakis, editors. *Semantic Search over the Web*. Springer Verlag, 2012.

[7] Roberto De Virgilio, Giorgio Orsi, Letizia Tanca, and Riccardo Torlone. Nyaya: A system supporting the uniform management of large sets of semantic data. In *ICDE*, pages 1309–1312, 2012.

[8] Wenfei Fan and Philip Bohannon. Information preserving xml schema embedding. *ACM Trans. Database Syst.*, 33(1), 2008.

[9] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3(1):264–275, 2010.

[10] Christiane Fellbaum, editor. *WordNet An Electronic Lexical Database*. The MIT Press, 1998.

[11] Brian Gallagher. Matching structure and semantics : A survey on graph-based pattern matching. *Artificial Intelligence*, pages 45–53, 2006.

[12] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[13] O. Hassanzadeh and M. P. Consens. Linked Movie Data Base (Triplification Challenge Report). In *I-SEMANTICS*, pages 194–196, 2008.

[14] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, pages 813–826, 2009.

[15] Arijit Khan, Nan Li, Xifeng Yan, Ziyu Guan, Supriyo Chakraborty, and Shu Tao. Neighborhood based fast graph search in large networks. In *SIGMOD Conference*, pages 901–912, 2011.

[16] Li Ma, Yang Yang, Zhaoming Qiu, Guo Tong Xie, Yue Pan, and Shengping Liu. Towards a complete owl ontology benchmark. In *ESWC*, pages 125–139, 2006.

[17] Thomas Neumann and Gerhard Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases. *PVLDB*, 3(1):256–263, 2010.

[18] Alexandra Poulovassilis and Peter T. Wood. Combining approximation and relaxation in semantic web path queries. In *ISWC*, pages 631–646, 2010.

[19] Alberto Sanfeliu and King-Sun Fu. A Distance measure between attributed relational graphs for pattern recognition. *IEEE transactions on systems, man, and cybernetics*, 13(3):353–362, 1983.

[20] Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *ICDE Conference*, pages 405–416, 2009.

[21] Peter T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[22] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.

[23] Shijie Zhang, Meng Hu, and Jiong Yang. Treepi: A novel graph indexing method. In *ICDE*, pages 966–975, 2007.

[24] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: distance index based subgraph matching in biological networks. In *EDBT*, pages 192–203, 2009.

[25] Shijie Zhang, Jiong Yang, and Wei Jin. Sapper: Subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1):1185–1194, 2010.

[26] Lei Zou, Lei Chen, and M. Tamer Özsu. Distance-join: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.