Distrib Parallel Databases
DOI 10.1007/s10619-014-7142-1

# Approximate querying of RDF graphs via path alignment

**Roberto De Virgilio · Antonio Maccioni · Riccardo Torlone**

© Springer Science+Business Media New York 2014

**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 a 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.

**Keywords** Path · Graph · RDF · Approximate matching · Alignment

## 1 Introduction

The Web 3.0 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

Communicated by Haixun Wang and Jeffrey Xu Yu.

R. De Virgilio (✉) · A. Maccioni · R. Torlone
Università Roma Tre, Rome, Italy
e-mail: dvr@dia.uniroma3.it; rde79@yahoo.com

A. Maccioni
e-mail: maccioni@dia.uniroma3.it

R. Torlone
e-mail: torlone@dia.uniroma3.it

Published online: 07 February 2014

technology 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 [1, 2]. Among them, a major problem lies in the difficulty for 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 [3–7] 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. A support to approximate query processing is particularly relevant in the context of linked data in which usually data do not strictly follow the ontology of reference and therefore queries posed against the schema may not retrieve valid answers.

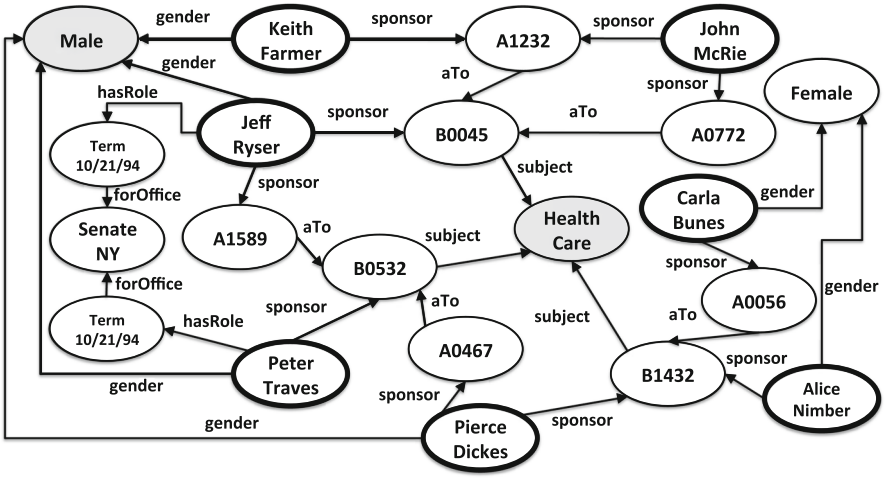
Since semantic data have a natural representation in the form of a graph, this problem has been often addressed in terms of approximate matching between a small graph $Q$ representing the query and a very large graph $G$ representing the database. The usual approach to this problem is based on searching all the subgraphs of $G$ that are isomorphic to $Q$. Unfortunately, this problem is known to be NP-complete [8] 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 [4,6,9], 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 [5]. Moreover, given that the set of answers to a query is potentially very large, a mechanism that aims to efficiently select the "best" $k$ answers is desirable.

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, thus avoiding the computation of all the candidate answers. We focus in particular on Basic Graph Pattern queries [7], which basically express conjunctive queries on graph data models, over RDF data. 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 values (also called literals), and links represent semantic relationships between resources. A resource is uniquely identified in the Semantic Web with a URI.

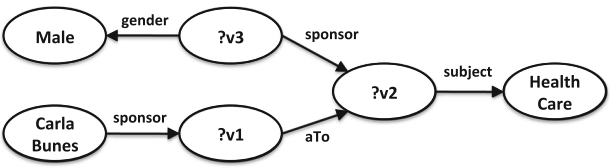
*Example 1* Let us consider the graph $G_d$ depicted in Fig. 1, taken from [4]: it represents a simplified portion of the GovTrack[1], a database that stores events that occurred in the US Congress. In RDF graphs, nodes represent RDF classes, literals, or URIs, whereas edges represent RDF properties.

Assume that a user needs to know all amendments sponsored by Carla Bunes to a bill on the subject of Health Care that was originally sponsored by a male person. Queries $Q_1$ and $Q_2$ in Fig. 1 are two possible ways to express this information need.

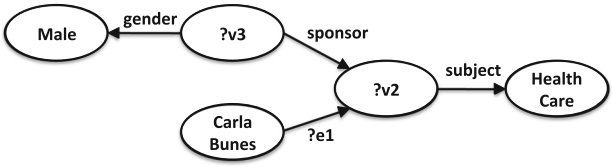
[1] http://www.govtrack.us.




**(a)** A RDF data graph $G_d$




**(b)** A query $Q_1$




**(c)** A query $Q_2$

**Fig. 1** An example of data and query graph

They only differ by the presence of an "optional" node and an "optional" edge. While $Q_1$ has an exact matching over $G_d$, a perfect matching algorithm would retrieve an empty result for $Q_2$ over $G_d$. Conversely, in the context of RDF data, it would be desirable to provide an answer also to $Q_2$.

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 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 $Q_2$ in Fig. 1, for which there is indeed no exact answer.

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 answers 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 polynomial 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 system[2] for querying RDF data that implements the above described technique. A number of experiments over widely used benchmarks have shown that our technique outperforms other approaches, in terms of both effectiveness and efficiency.

The rest of the paper is organized as follows. In Sect. 2 we introduce some preliminary notions and definitions. In Sect. 3 we illustrate our strategies for graph matching over RDF data and in Sect. 5 we present the experimental results. In Sect. 6 we discuss related work and finally, in Sect. 7, we draw some conclusions and sketch some future work.

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

## 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 an 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.

### 2.1 Problem definition

A graph is a 4-tuple $G = \langle N, E, L_N, L_E \rangle$ 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 injective functions that associate an element of a set of *node labels* $\Sigma_N$ to each node in $N$ and an element of a set of *edge labels* $\Sigma_E$ to each edge in $E$, respectively.

We focus our attention on (possibly large) RDF databases, which are conceptually conceived as labeled directed graphs in which the nodes represent either resources or values while the edges relate resources to resources and resources to values. We then introduce the following notion. Let $\mathcal{U}$ be a set of *URIs* and $\mathcal{L}$ be a set of *literals*.

**Definition 1** (*Data Graph*) A data graph $Q$ is a graph where $\Sigma_N = \mathcal{U} \cup \mathcal{L}$ and $\Sigma_E = \mathcal{U}$.

Let VAR be a set of *variables*, denoted by the prefix "?". A *query graph* $Q$ is a data graph in which the nodes can be labeled with variables.

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

The evaluation of a query consists on retrieving portions of the data graph that match with the query graph. This process can be relaxed by assuming that before the match, the query graph can be slightly transformed, as formalized in the following.

A *substitution* for a query graph $Q$ is a function that maps the variables in $Q$ to either URIs or literals. A *transformation* $\tau$ 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 $\phi$ and a transformation $\tau$ such that $G' = \tau(\phi(Q))$. If $\tau$ is the identity function, $G'$ is an *exact* answer to $Q$.

In our implementation, the operation of labeling modification of the $\tau$ function is based on standard libraries for testing the equality of values based on traditional techniques for full text search (such as stemming). This allows the matching between labels such as *fishing*, *fished*, and *fish*. Since this aspect is outside the scope of the paper, we will simply assume, hereinafter, that the operation of labeling modification provides a support for approximate matching between labels and we will use the term *matching* between values in this sense, without discussing this aspect further.

Intuitively, an answer $a_1 = \tau_1(\phi_1(Q))$ is more relevant than another answer $a_2 = \tau_2(\phi_2(Q))$ if $\tau_1$ contains a lower number of operations than $\tau_2$. 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 could be reasonable that the modification of a label is less relevant than a node insertion, since the latter increases the semantic distance between concepts. Therefore, let $\omega$ be a function that associates a *weight of relevance* to each basic operation $\odot$. We say that the *cost* $\gamma$ 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.

## 2.2 Paths and computed answers

We now introduce a number of notions that, in our approach, are used in the construction of the answers to a query. Given a graph $G$, we call *start* nodes, the nodes of $G$ with no in-going edges, and *end* nodes, the nodes of $G$ with no out-going edges.

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

**Definition 5** (*Path*) Given a data graph $G = \langle N, E, L_N, L_E \rangle$, a path is a sequence $p = l_{n_1} - l_{e_1} - l_{n_2} - \cdots - l_{e_{k-1}} - l_{n_k}$ where: (i) $l_{n_i} = L_N(n_i)$, $l_{e_i} = L_E(e_i)$, and $n_i \in N$, $e_i = (n_i, n_{i+1}) \in E$, (ii) $n_1$ is either a start node or, if $G$ has no start nodes, a hub, and (iii) $n_k$ is either an end node or a node such that there is no edge $(n_k, n_{k+1})$ such that $L_N(n_{k+1})$ (the label of $n_k$) already occurs in $p$.

In the following, give a path $p = l_{n_1} - \cdots - l_{n_k}$, we will call $n_1$ and $n_k$ the *source* and the *sink* of $p$, 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 among the nodes in the path.

Let us consider for instance the data graph $G$ in Fig. 2. This graph has two loops, no start nodes, no end nodes, and one hub ($n_2$). It follows that the paths of $G$ are: $p_1 = n_2 - e_2 - n_1$, of length 2, and $p_2 = n_2 - e_3 - n_3 - e_4 - n_1$, of length 3.

Let us now consider the data graph in Fig. 1. It has seven start nodes (the double-marked nodes) and two end nodes (*Health Care* and *Male*, marked in gray). An example of path is:

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

where JR and HC denote *Jeff Ryser* and *Health Care*, respectively. $p_z$ has length 4 and the node A1589 has position 2. The query $Q_1$ in Fig. 1 has the following paths:

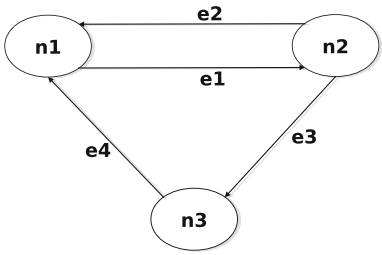



**Fig. 2** A data graph with loops and without start nodes

$q_1$ : 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 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 $\phi$ and a transformation $\tau$ of a path $p$ of $Q$ such that $\tau(\phi(p))$ is a path of $G$.

We are ready to introduce our notion of *computed* answer. 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** (*Computed Answer*) Given a query graph $Q$, a computed answer of $Q$ over a data graph $G$ is a set of alignments of all the paths of $Q$ that forms a connected component of $G$.

Note that a computed answer of $Q$ over a data graph $G$ is indeed an answer of $Q$ over $G$ (Definition 3). For this reason, in the following we will often not make any distinction between answer and computed answer, when it is clear from the context the notion we are referring to.

## 2.3 Scoring function

The function $score$ is an approximate implementation of the general notion of relevance (Definition 4) that can be computed in linear time on the size of the data. The function $score$ simulates the relevance of computed answers $a_i$ by taking into account two different aspects, *quality* and *conformity*. The former measures how much the paths retrieved align with the paths in the query. The latter measures how much, in $a_i$, the combination of paths retrieved is similar to the combination of the paths in the query.

The first aspect that $score$ considers is the quality of alignment between paths of a computed answer $a_i$ and paths of a query $Q$ as follows:

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

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

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

In this expression: (i) $n_N^-$ and $n_E^-$ are, respectively, the number of nodes and edges of $p$ that are not present in $q$, and (ii) $n_N^{\curvearrowright}$ and $n_E^{\curvearrowright}$ are, respectively, the number of nodes and edges inserted in $q$ by $\tau$. Finally, $a$, $b$, $c$ and $d$ are parameters that serve to take into account the weights of relevance of the operations in $\tau$ (see Definition 4).

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

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

In this equation, $q_i$ and $q_j$ are paths of $Q$, $p_i$ and $p_j$ are paths of $a_i$ 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 $\psi$ is a function defined as follows:

$$\psi(q_i, q_j, p_i, p_j) = \begin{cases} e \cdot \frac{|\chi(q_i, q_j)|}{|\chi(p_i, p_j)|}, & \text{if } |\chi(p_i, p_j)| > 0 \\ e \cdot |\chi(q_i, q_j)|, & \text{if } |\chi(p_i, p_j)| = 0 \end{cases}$$

where $\chi$ 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 $\chi(q_i, q_j)$. Finally, $e$ is a parameter that serves to take into account the weight of the conformity in $score$. This is very useful in those applications where the topology and the interconnections within the answers are very important (e.g., in social network analysis), sometimes even more than the content of the data themselves.

The final score function is then computed as:

$$score(a_i, Q) = \Lambda(a_i, Q) + \Psi(a_i, Q)$$

It turns out that, with a suitable choice of the parameters in Eq. (1) that considers the weights of relevance assigned to the basic operations, *score* is *coherent* with the notion of relevance of an answer, that is, for each pair of answers $a_1$ and $a_2$ for a query $Q$ such that $a_1$ is more relevant than $a_2$ we have that $score(a_1, Q) < score(a_2, Q)$.

**Theorem 1** *Given a query graph $Q$ and a data graph $G$, for each pair of computed answers $a_i$ and $a_j$ for $Q$ over $G$, if $a_i$ is more relevant than $a_j$ then we have $score(a_i, Q) < score(a_j, Q)$.*

*Proof* Let $\odot_N^{\curvearrowright}$, $\odot_N^-$ and $\odot_N^\times$ be basic update operations of node insertion, node deletion, and labeling modification, respectively. Analogously, $\odot_E^-$, $\odot_E^{\curvearrowright}$ and $\odot_E^\times$ are the respective operations on edges. On these operations, we fix the function $\omega$: (i) $\omega(\odot_N^-) = a$, (ii) $\omega(\odot_N^{\curvearrowright}) = b$, (iii) $\omega(\odot_E^-) = c$ and (iv) $\omega(\odot_E^{\curvearrowright}) = d$. We consider, as in other works [10], $\omega(\odot_N^\times) = 0$ and $\omega(\odot_E^\times) = 0$ because we do not 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 $\tau_i$ for an answer $a_i$. In this case: $n_N^-$ and $n_E^-$ are, respectively, the number of nodes and edges of $a_i$ that are inserted in $Q$, and $n_N^{\curvearrowright}$ and $n_E^{\curvearrowright}$ are, respectively, the number of nodes and edges updated in $Q$ by $\tau_i$.

The cost of $\gamma(\tau_i)$ is $n_N^- \cdot a + n_N^{\curvearrowright} \cdot b + n_E^- \cdot c + n_E^{\curvearrowright} \cdot d$. Let $a_1 = \tau_1(\phi_1(Q))$ and $a_2 = \tau_2(\phi_2(Q))$ be two answers over a query $Q$. 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

$$\gamma(\tau_1) < \gamma(\tau_2) \tag{2}$$

But for a path $p \in a_i$ we have that $\gamma(\tau_i) = \lambda(p, Q)$. Then, if we generalize Eq. (2) to all the paths of the two answers $a_1$ and $a_2$ we obtain

$$\Lambda(a_1, Q) < \Lambda(a_2, Q) \tag{3}$$

that satisfies the hypothesis for the first aspect of $score$. With $score$ we have an upper bound of the notion of relevance because if a node has a mismatch in $Q$ and it is in common among more than one path, then it gets counted more than once in $n_N^-$ and $n_N^{\curvearrowright}$. The conformity $\Psi(a_i, Q)$ follows a similar trend than $\Lambda(a_i, Q)$. In fact, more are the mismatching elements (i.e. nodes and edges) and lower is the number of common nodes. Consequently, the number of intersections between the paths in an answer conforming to the intersections between the paths of the query is also lower. In our case we have that

$$\Psi(a_1, Q) < \Psi(a_2, Q). \tag{4}$$

Given Eqs. (3) and (4), it follows that $score(a_1, Q) < score(a_2, Q)$. □

## 2.4 Computation of alignments

In order to measure $\Lambda$ in $score$ we have to compute alignments between paths in $a$ and paths in $Q$. Therefore, $a$ and $Q$ are first decomposed into a set of paths. Then, the paths of $a$ are aligned against paths of $Q$. In our example, this method would decompose $Q_1$ in the following paths:

$$q_1 : \text{Carla Bunes} \xrightarrow{sponsor} \text{?v1} \xrightarrow{aTo} \text{?v2} \xrightarrow{subject} \text{Health Care}$$
$$q_2 : \text{?v3} \xrightarrow{sponsor} \text{?v2} \xrightarrow{subject} \text{Health Care}$$
$$q_3 : \text{?v3} \xrightarrow{gender} \text{Male}$$

Now assume that $a_1$ is extracted from $G_d$ that is formed by the following paths:

$$p_1 : \text{Carla Bunes} \xrightarrow{sponsor} \text{A0056} \xrightarrow{aTo} \text{B1432} \xrightarrow{subject} \text{Health Care}$$
$$p_2 : \text{Pierce Dickens} \xrightarrow{sponsor} \text{B1432} \xrightarrow{subject} \text{Health Care}$$
$$p_3 : \text{Pierce Dickens} \xrightarrow{gender} \text{Male}$$

Note that in the example above the result is an exact answer to $Q_1$, but the same strategy can be adopted to compare paths that are not exactly matches and thus, with a lower degree of similarity.

We can now compute the quality of alignments between the paths $p$ in $a_1$ and the paths $q$ in $Q$. They are done by inserting, deleting and modifying nodes in $q$ by proceeding with a scan contrary to the direction of the edges. For instance let us consider $q_1$ and $q_2$ from the query graph $Q_1$ and the 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

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

In this case $q_1$ requires only a substitution $\phi$ on the variables while $q_2$ employs a transformation $\tau$ to insert **aTo-B1432** and a substitution $\phi$ on the variables. In the former case we have $\lambda(p, q_1) = (0+0)+(0+0) = 0$, since $n_N^- = n_N^\curvearrowright = n_E^- = n_E^\curvearrowright = 0$. In the latter case $\lambda(p, q_2) = (0+b)+(0+d)$, since $n_N^- = n_E^- = 0$, and $n_N^\curvearrowright = n_E^\curvearrowright = 1$. If we set $b = 0.5$ and $d = 1$, we have $\lambda(p, q_2) = 1.5$ (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 have $\lambda(p', q_1) = (a+0)+(0+0)$, since $n_V^- = 1$ due to the mismatch between CB and JR. If we set $a = 1$, we have that $\lambda(p', q_1) = 1$ (i.e. $q_1$ has a better alignment with $p$ than $p'$).

It is straightforward to demonstrate that the time complexity of the alignment is $O(I)$ where $I = |p| + |q|$ is the sum of the nodes and edges of the paths in $p$ and $q$.

## 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 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 Sect. 5.

In the second phase, the set PD of the paths of $Q$ are first retrieved by exploiting the index and then PD is used to retrieve the best answers by adopting a strategy that guarantees a polynomial time complexity with respect to the size of PD. This task is performed by means of the following main steps:

*Preprocessing* Given a query graph $Q$, in this step the set PQ of all paths of $Q$ is computed on the fly by traversing $Q$. 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* ($IG$). Nodes of $IG$ are 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 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 Fig. 3. This data structure keeps track of the fact that $q_1$ and $q_2$ have nodes in common (?v2 and Health Care) and that $q_2$ and $q_3$ have also nodes in common (only ?v3).
*Clustering* In the second step we build a cluster for each element $q$ of PQ. Then, we group in the same cluster all the paths $p$ of $G$ having a sink that matches the sink of $q$. If a variable occurs in the sink of $q$, we retrieve the last value $v$ occurring in $q$ and we group in the same cluster all the paths $p$ of $G$ containing a label matching $v$. Before the insertion of a 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 paths in a clusters are ordered according to their score, with the lower coming first. Note that the same 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 Fig. 1, we obtain the clusters shown in Fig. 4. In this case clusters $cl_1$, $cl_2$ and $cl_3$ correspond to the paths $q_1$, $q_2$ and $q_3$ of PQ, respectively; note the scores at the right side of each path and in particular the path $p_1$ occurring in both $cl_1$ and $cl_2$ with different scores, i.e. 7 in $cl_1$ and 5 in $cl_2$.


**Fig. 3** An example of intersection query graph

$$
cl_1 : \begin{pmatrix} p_1 : \text{CB-sponsor-A0056-aTo-B1432-subject-HC } [0] \\ p_2 : \text{JR-sponsor-A1589-aTo-B0532-subject-HC } [1] \\ p_3 : \text{KF-sponsor-A1232-aTo-B0045-subject-HC } [1] \\ p_4 : \text{JM-sponsor-A0772-aTo-B0045-subject-HC } [1] \\ p_5 : \text{JM-sponsor-A1232-aTo-B0045-subject-HC } [1] \\ p_6 : \text{PD-sponsor-A0467-aTo-B0532-subject-HC } [1] \end{pmatrix}
$$

$$
cl_2 : \begin{pmatrix} p_7 : \text{JR-sponsor-B0045-subject-HC } [0] \\ p_8 : \text{PT-sponsor-B0532-subject-HC } [0] \\ p_9 : \text{AN-sponsor-B1432-subject-HC } [0] \\ p_{10} : \text{PD-sponsor-B1432-subject-HC } [0] \\ p_{11} : \text{CB-sponsor-A0056-aTo-B1432-subject-HC } [1.5] \\ p_{12} : \text{JR-sponsor-A1589-aTo-B0532-subject-HC } [1.5] \\ p_{13} : \text{KF-sponsor-A1232-aTo-B0045-subject-HC } [1.5] \\ p_{14} : \text{JM-sponsor-A0772-aTo-B0045-subject-HC } [1.5] \\ p_{15} : \text{JM-sponsor-A1232-aTo-B0045-subject-HC } [1.5] \\ p_{16} : \text{PD-sponsor-A0467-aTo-B0532-subject-HC } [1.5] \end{pmatrix}
$$

$$
cl_3 : \begin{pmatrix} p_{17} : \text{JR-gender-Male } [0] \\ p_{18} : \text{KF-gender-Male } [0] \\ p_{19} : \text{JM-gender-Male } [0] \\ p_{20} : \text{PD-gender-Male } [0] \end{pmatrix}
$$

**Fig. 4** An example of the clustering step

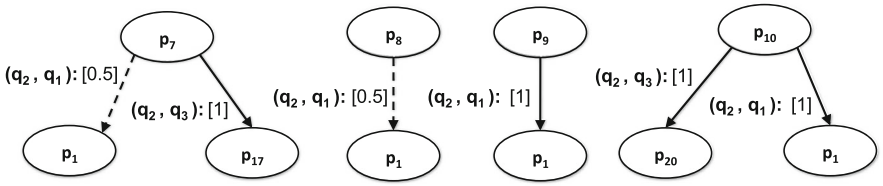



**Fig. 5** Forest of paths

*Search* The last step aims at generating the most relevant answers by combining the paths in the clusters built in the previous step. This is done by picking and combining the paths with lowest score from each cluster. The intersection query graph allows us to verify efficiently if they form an answer. As an example, given the clusters in Fig. 4, the first answer is obtained by combining the 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 paths of PQ associated with the clusters.

The most tricky task of the whole process occurs in the third step above, where we try to generate the top-k answers by minimizing the number of combinations between paths. This is done by organizing the combinations of paths in a forest where nodes represent the retrieved paths, while edges between 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 paths corresponding to the clusters where $p_i$ and $p_j$ were included, respectively.

For instance, Fig. 5 reports the forest for the paths with the higher score extracted from the clusters in Fig. 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 answer.

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 paths from $G$ ending into the sinks of the paths of PQ, as shown in Algorithm 1.

**Algorithm 1**: Clustering of paths

**Input** : The query 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 ;
```

The set $\mathcal{CL}$ of clusters is implemented as a map where the key is a path $q$ from PQ and the value is a cluster with all the paths $p$ ending in the sink of $q$. Each cluster is implemented as a priority queue of paths, where the priority is based on the score associated with each path (in ascending order). In this paper, we use an implementation of priority queues to guarantee a constant 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 $G$ matching $sk$. Such task is performed by the function getPaths [line 3]. Once we have 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].

### 3.3 Search

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

As long as we did not generate $k$ answers and the set of clusters is not empty [line 1], we build a forest $\mathcal{F}$ [line 2] from the most promising paths in $\mathcal{CL}$ and we provide the top-k answers by visiting $\mathcal{F}$ [lines 4–8]. As we have said in Sect. 3.1, nodes of $\mathcal{F}$ denote paths in $\mathcal{CL}$, while edges $(p_i, p_j)$ represent the fact that $p_i$ and $p_j$ have nodes

**Algorithm 2**: Search

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

```
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 ← roots − {r};
8          a ← Visit(r, getQPath(CL, r ), V, F.get(r ));
9          S.add(a );
10 return S ;
```

in common. The trees of $\mathcal{F}$ are then returned as answers to the query. In the following we describe in more detail the building and the visiting of $\mathcal{F}$.

*Building* $\mathcal{CL}$ and the intersection query graph $IG$, first of all we have a *building* phase generating a *forest* of 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 ;
```

$IG$ is used to evaluate the conformity of the answers 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 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 4]. Referring to our example the cluster $cl_2$ corresponds to $q_2$. In this case the top paths to dequeue are $p_7$, $p_8$, $p_9$ and $p_{10}$. The 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 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 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 $p \in \mathsf{PD}$, 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 path $q$, intersection query graph $IG$, visited V.

```
1  L ← ∅;
2  foreach (q, q′) ∈ IG.E : q′ ∉ V do
3      cl ← CL.get(q′);
4      PD ← cl.dequeueTop();
5      foreach p ∈ PD : p ↔ r do
6          T.V ← T.V ∪ {p};
7          e ← ⟨r, p, (q, q′) : [ψ(q, q′, r, p)]⟩;
8          T.E ← T.E ∪ {e};
9          L ← L ∪ {⟨p, q′⟩};
10     V ← V ∪ {q′};
11 foreach ⟨p, q′⟩ ∈ L do
12     treeBuild(p, T, CL, q′, IG, V);
```

The procedure treeBuild starts the navigation of $IG$ from the input query path $q$. For each edge $(q, q')$, where $q'$ is not yet visited, we dequeue the top paths PD from the cluster cl corresponding to $q'$ [lines 3–4]. Then for each path $p \in \mathsf{PD}$ having nodes in common with r (denoted by $p \leftrightarrow \mathsf{r}$), we build the edge (r, $p$) and the corresponding label $\langle \mathsf{r}, p, (q, q') : [\psi(q, q', \mathsf{r}, p)]\rangle$, [line 7], and we insert it in the set $E$ of T [line 8]. The pair $\langle p, q'\rangle$ is added to the set L [line 9]. Finally, we include $q'$ in the set V of visited query paths [line 10] and we recursively call the procedure for each $p$ [line 12].

As an example, Fig. 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

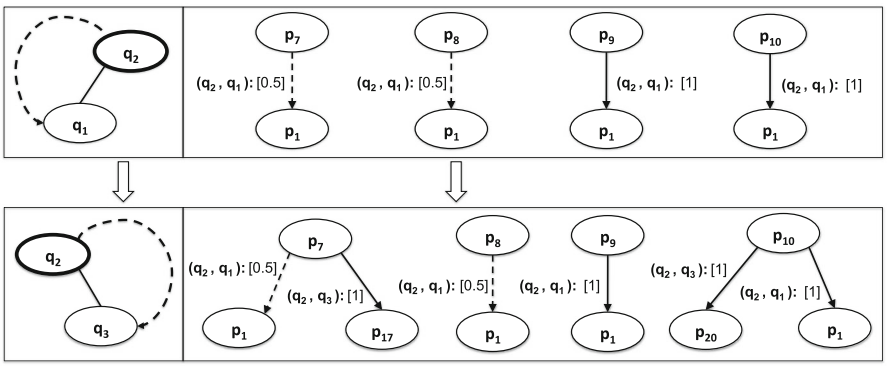



**Fig. 6** Building of the forest

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 it has 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. Algorithm 3 requires $I$ iterations, where $I$ is the number of paths PD retrieved in line 3, which, in the worst case, are all the paths of the data graph. The most involved operation in the body of the loop in Algorithm 3 is a call to Algorithm 4 whose goal is the construction of a tree $T$, where the nodes represent paths of the data graph $G$ and the edges connect paths having a non-empty intersection. The algorithm is made of two parts (lines [2–10] and [11–12]). In the first part there is a nested loop. The external loop iterates over the query paths in the query graph, which are at most $h$. The internal loop iterates over the paths in the data graph that align with the paths in the query, which are at most $I$. It follows that the first part requires, at most, $h \times I$ steps. At each step, a pair $\langle p, q' \rangle$, where $p$ is a path of the data graph that aligns with the path $q'$ in the query graph, is added to the set L. The second part of the algorithm iterates over the elements of L, which are at most $I$. In each iteration, there is a recursive call to the algorithm. In the base case, the query graph consists in just one path and the cost is constant since no iteration is required both in the first and in the second part of the algorithm. In the general case, since at each call of the algorithm we consider only the paths of the query graph that have not been visited yet (line 2), we have that the cost $C$ of the algorithm can be expressed as: $C(h, I) = (I \times h) + I \times C(h - 1, I)$. Therefore, we have that the complexity of Algorithm 4 is the solution of the following equation:

$$C(h, I) = \begin{cases} I \times C(h-1, I) + I \times h, & h > 1 \\ 1, & h = 1 \end{cases}$$

From this equation, it follows that the complexity of Algorithm 4 is in $O(I^{h-1})$ and since Algorithm 3 requires $I$ iterations of Algorithm 4, the overall complexity of Algorithm 3 is in $O(I \times I^{h-1}) \in O(I^h)$.

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

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 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 path $p'$ with the getPathMaxConformity [line 5]. We then include $p'$ in the answer a and we call recursively the algorithm [line 7]. If $p$ has no successors, then we return $p$.

**Algorithm 5**: Visit

**Input** : path $p$, path $q$, visited $\mathsf{V}$, tree $\mathsf{T}$.
**Output**: set of paths $\mathsf{a}$.

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

**Table 1** Overall complexity

| Clustering | Building | Visiting | Search | Overall |
|---|---|---|---|---|
| $\|Q\| \times O(I)$ | $O(I^h)$ | $O(h^{h-1} \times I)$ | $O(k \times I^h)$ | $O(k \times I^h)$ |

Referring to Fig. 6, we start from roots = $\{p_{10}, p_7, p_9, p_8\}$ (i.e. in order of priority). Then the first answer $a_1$ dequeues $p_{10}$. From $p_{10}$, we add $p_{20}$ and $p_1$ to $a_1$, that are the most important paths to which $q_3$ and $q_1$ are associated. Finally, we have $a_1 = \{p_{10}, p_1, p_{20}\}$. Similarly we generate, in order, $a_2 = \{p_7, p_1, p_{17}\}, a_3 = \{p_9, p_1\}$ and $a_4 = \{p_8, p_1\}$.

In the base case of Algorithm 5 [lines 1–2] either the query graph $\mathsf{T}$ has one node only. In this case, the cost is constant since no iteration is performed. In the general case, as for the building, the visit explores the graph $IG$ and iterates at most $h$ times [lines 4–7]. In each iteration there is a call to function getPathMaxConformity, which has a cost in $O(I)$ since it checks the conformity of all the children of $p$ in $\mathsf{T}$, which are at most $I$. Then, there is a recursive call to the algorithm and so, since at each call we consider only the paths of the query graph that have not been visited yet [lines 6–7], the cost can be expressed, in the general case, as $C(h, I) = h \times (I + C(h - 1, I))$. Therefore, the computational time complexity of Algorithm 5 is the solution of the following equation:

$$C(h, I) = \begin{cases} h \times (I + C(h-1, I)), & h > 1 \\ 1, & h = 1 \end{cases}$$

From this equation, it follows that the complexity of Algorithm 5 is in $O(h^{h-1} \times I)$.

*Overall complexity* Table 1 summarizes all the discussed complexity. The overall process consists of two main phases: clustering (Algorithm 1) and search (Algorithm 2). In turn, the search algorithm consists of two main parts: building (Algorithm 3) and visiting (Algorithm 5). As we have said above, search is the core of the overall process and iterates at most $k$ times, where $k$ is the number of answers to return. In each iteration, there is a call to building and, at most, $I$ calls to visiting. Therefore $O(\text{search}) \in k \times O(\text{building}) + k \times I \times O(\text{visiting}) \in O(k \times I^h)$, where $I$ is the number of paths retrieved and $h$ is the number of paths in $Q$. It turns out that the overall complexity is bounded by the complexity of the search phase. Note that the

complexity is exponential in the size of the *query*, which is usually very limited with respect to the size of data. On the other hand, we have that the maximum number of paths in a data graph $G$ (see Definition 5) is proportional to $n^2$ where $n$ is the number of nodes of $G$. This is because, in the worst case, we have $n/2$ sources and $n/2$ sinks with edges between each source and each sink, and so we have $n^2/4$ paths. It follows that our technique exhibits a polynomial time complexity in the size of the *data*. Note that, as soon as $I$ tends to $n^2$, the depth of the tree computed in the build phase tends to 1, because the graph becomes strongly connected. In this case, the complexity of search reduces to $O(k \times I)$. We also point out that, in Sect. 5 we show that our technique improves other approaches in terms of efficiency over real world data sets and it scales seamlessly with the size of the input.

## 4 Implementation

We have implemented our approach in SAMA[3], a Java system with a Web front end.

To build answers efficiently, we index the following information: vertices' and edges' labels of the data graph $G$ (for element-to-element mapping) and the 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 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 BFS paradigm, where independently concurrent traversals are started from each source. Similarly to [4], and differently from the majority of related works (e.g., [5]), we assume that the graph cannot fit in memory and that can only be stored on disk. Such algorithm allows to retrieve a polynomial number of paths; of course it is not the complete set of paths between sources and sinks, but as shown in our experimentations such set allows a good effectiveness. Specifically, we store the index in a GraphDB, that is HyperGraphDB [11] (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 indexed by an index $m \in M$ and $f \in F$, respectively. Figure 7 shows an example of reference.

Physically HGDB implements several `HGHandle` to wrap and to index nodes and edges of $G$; each subset of nodes (i.e. hyperedge) is implemented into a `HGEdgeLink` having several *target* links to the `HGHandle` of contained nodes (i.e. a hyperedge is implemented as a list of cursors to the contained nodes). In our framework, each path is modeled as a `HGEdgeLink` in HGDB. The matching is supported by standard IR

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

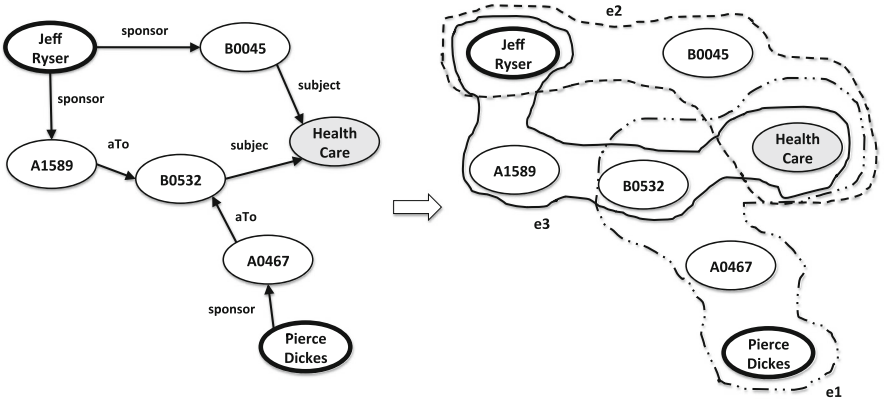



**Fig. 7** An example to represent a data graph $G$ (*left side*) in a hypergraph $H$ (*right side*)

engines (c.f. Lucene Domain index—LDi[4]) 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 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 [12], supported by LDi.

## 5 Experimental results

In our experiments we used several widely-accepted benchmarks for graph matching evaluation. We have compared SAMA with three representatives graph matching systems: SAPPER [6], BOUNDED [5] and DOGMA [4]. Experiments were conducted on a dual core 2.66 GHz Intel Xeon, running Linux RedHat, with 4 GB of memory and a 2-disk 1Tbyte striped RAID array.

*Indexing* In our experiments we consider real RDF datasets, such as PBLOG[5], GOVTRACK, KEGG, IMDB [13], DBLP, and synthetic datasets, such as BERLIN [14], LUBM [15] and UOBM [16]. Table 2 provides importing information for any dataset: number of triples, number of nodes ($|HV|$ column) and number of generated hyperedges ($|HE|$ column) in HGDB, time to create the index on HGDB (t column) 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 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, given a label, a path $p$ and all data elements (nodes and edges) associated with $p$. We performed *cold-cache* experiments (i.e. by dropping all file-system caches before restarting the

[4] http://lucene.apache.org/.

[5] http://www-personal.umich.edu/~mejn/netdata/.

**Table 2** HyperGraphDB indexing

| DG | #Triples | $\|HV\|$ | $\|HE\|$ | t | Space |
|---|---|---|---|---|---|
| PBlog | 50 K | 1,5 K | 96 K | 1 sec | 56 MB |
| GOV | 1 M | 280 K | 330 K | 4 min | 340 MB |
| KEGG | 1 M | 300 K | 606 K | 7 min | 700 MB |
| Berlin | 1 M | 320 K | 700 K | 10 min | 910 MB |
| iMDB | 6 M | 900 K | 3 M | 47 min | 1.2 GB |
| LUBM | 12 M | 1 M | 15 M | 102 min | 12.9 GB |
| UOBM | 12 M | 1 M | 15 M | 102 min | 12.9 GB |
| DBLP | 26 M | 4 M | 17 M | 441 min | 23.6 GB |

**Table 3** Average time to retrieve a path

| DG | Cold (msec) | Warm (msec) |
|---|---|---|
| IMDB | 0.01 | 0.005 |
| LUBM | 0.04 | 0.008 |
| DBLP | 0.06 | 0.009 |

**Table 4** Index maintenance performance

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

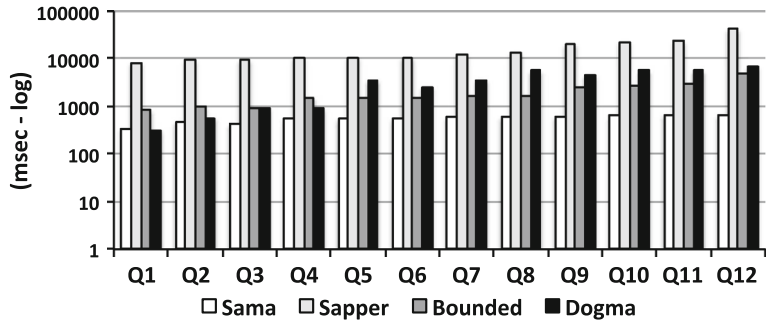
various systems and running the queries) and *warm-cache* experiments (i.e. without dropping the caches).

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 $G$. Such operations are documented in [17].

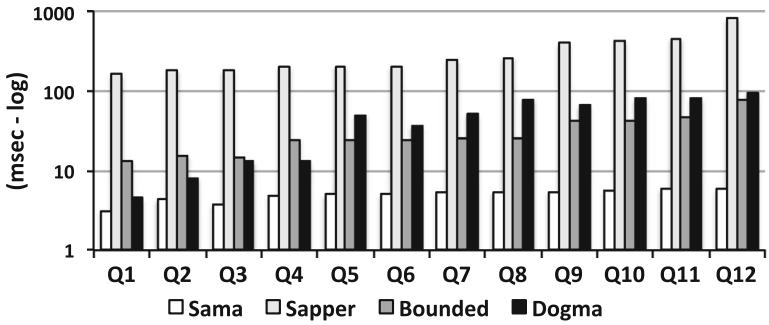
Update operations perform well, as demonstrated in Table 4. We inserted and updated 100 vertices (edges) and then we deleted them. We measured the average response time (ms) to insert/delete/update one vertex (edge). Once the index is built the first time, it can be maintained easily as the maintenance 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). In the following we refer to the most huge datasets: in particular we will discuss in depth results over LUBM dataset, that are the most representative. DBLP and iMDB provide a very close behavior. Hence, we consider 12 queries from the LUBM benchmark that provide results without involving reasoning.[6] 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

[6] At https://www.dropbox.com/sh/d5u1u24qnyqg18f/7Oefq8-qVa you can find the complete set of queries.




**(a)** cold-cache




**(b)** warm-cache

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

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 $d = 1$. Therefore we show the behavior of all systems in terms of number of triples and complexity. The query run-times are shown in Fig. 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 Fig. 9 shows 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 paths, and to retrieve all paths. Of course a decentralized environment could relevantly limit this time consumption. However time consumption percentages of the main steps show 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.

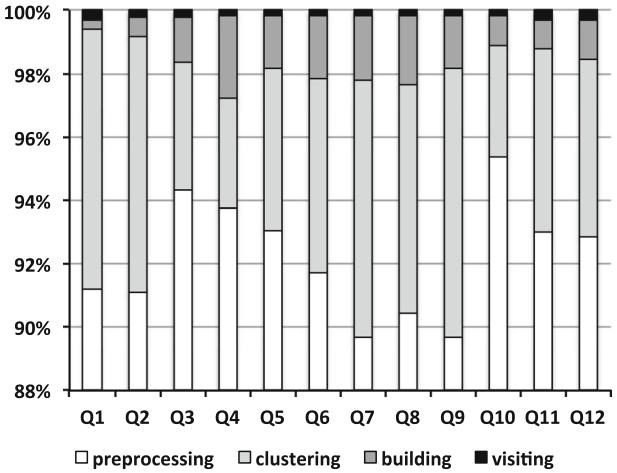



**Fig. 9** Cumulative time percentage of each step

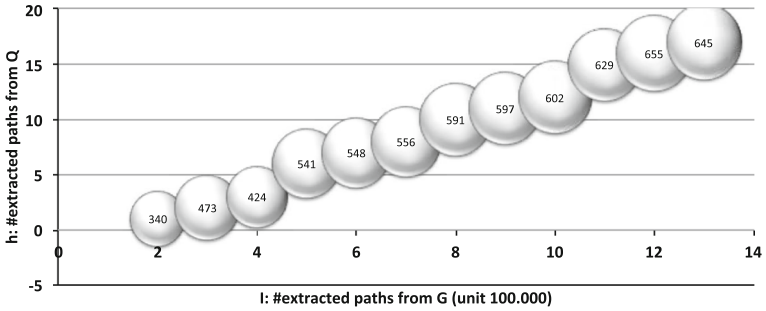



**Fig. 10** Flexibility of SAMA on LUBM

Another aspect that we test is the scalability of our approach. In particular this experiment shows that the time of execution is quadratic with respect to size of data and this is coherent with the polynomial time complexity of our technique (see Sect. 3). Figure 10 shows the flexibility of SAMA with respect to both the number $I$ of paths extracted from $G$ and the number $h$ of paths extracted from $Q$. For each couple $(I, h)$ we depict the average response time (in 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 $h$: 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 to $I$, $h$, the number of nodes and the number of variables in $Q$, through distinct diagrams, as illustrated in Fig. 11 (i.e. it refers to cold-cache experiments). The diagram displays the trend-line and the interpolated equation: in any case, the behavior of SAMA is polynomial with respect to the size of data; in particular the trend-line always approximate a quadratic trend (i.e. we have the same for warm-cache experiments).

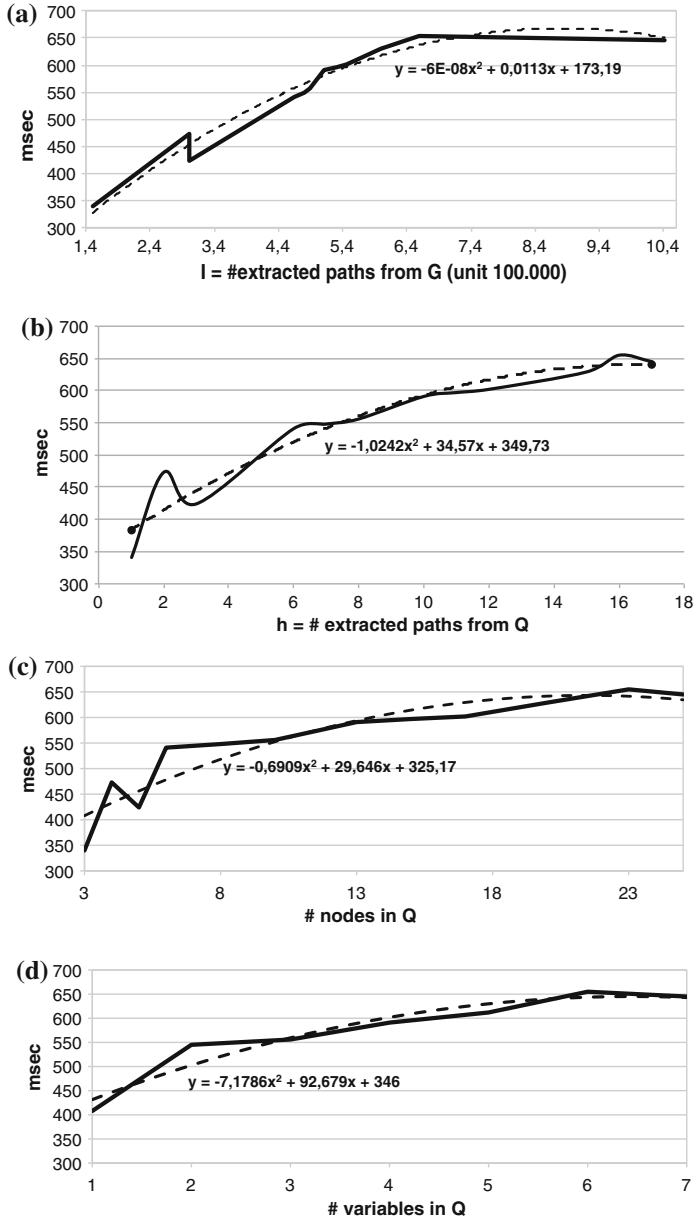



**Fig. 11** Scalability of SAMA on LUBM with respect to **a** the number $I$ of extracted paths from $G$, **b** the number $h$ of extracted paths from $Q$, **c** the number of nodes in $Q$ and **d** the number of variables in $Q$

*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

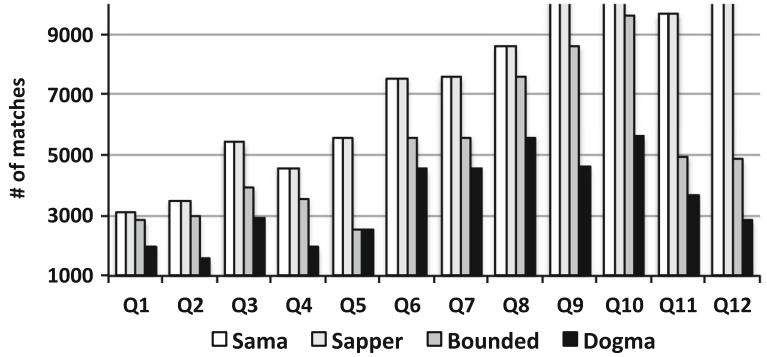



**Fig. 12** Effectiveness on LUBM: *bars* refer each system, using different *gray scales*, i.e. from SAMA, *white bars*, to DOGMA, *black bars*

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 answers returned. Figure 12 shows the effectiveness of all systems on LUBM, where we run the queries without imposing the number k of answers.

In this case SAMA and SAPPER always identify more meaningful matches than both 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 meaningfulness of results, we measured the interpolation between precision and recall, 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. $P(r_j) = max_{r_j \leq r \leq r_{j+1}} P(r)$. Figure 13 shows the results on LUBM: for SAMA we depict three different trends with respect to the range of $|Q|$. As is to be expected, queries with limited number of 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.

## 6 Related work

Many research efforts have focused on graph similarities, specially from the field of graph matching [8]. In fact, a first category of works relies on subgraph isomorphism [18]. However the well-known intractability of the problem inspired approximate approaches to simplify the problem [8,19]. In particular, graph simulation techniques has been used to make graph matching tractable. 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

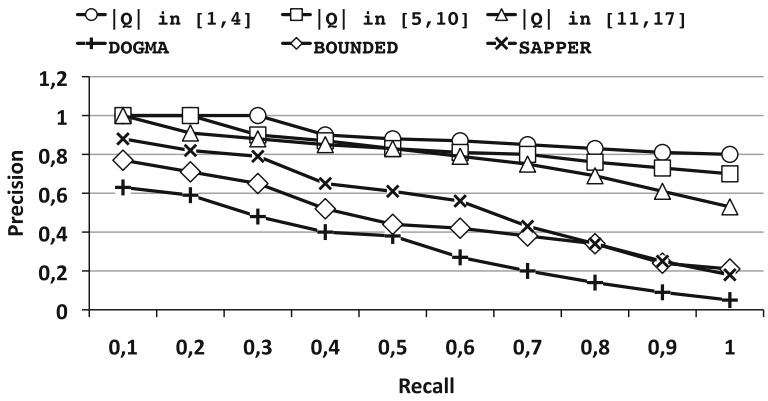



**Fig. 13** Effectiveness on LUBM: precision and recall of SAMA

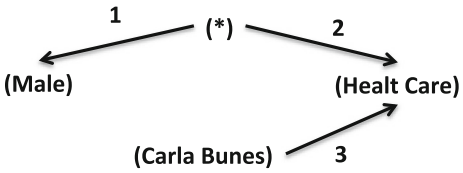



**Fig. 14** An example of bounded query

others have proposed specific indexes for the efficient execution of SPARQL queries and joins [21]. 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 [24], 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 [4], TALE [25], GADDI [9], SAPPER [6], and Zeng et al. [26] 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 [27,28] and distance queries [29] based on testing the existence of a path between two nodes in the graph and on the evaluation of the distance between them. An interesting approach is proposed in [5] 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. For instance, we can represent $Q_1$ of Example 1 in terms of a bounded query as shown in Fig. 6. This guarantees a cubic time complexity for the graph matching problem (Fig. 14).

Most of the mentioned works are focused on medical, chemical and proteinic networks and they are usually not efficient over semantic and social data [10]. Therefore, specialized metrics where proposed [10,30]. GMO [30] introduces a structural metric based on a bipartite graph, NESS [10] proposes a measure based on both topological and content information in the neighborhood of a node of the graph. 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 have to be correlated. It relies on the tractable problem of alignment between paths.

## 7 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 polynomial 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 answers 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. De Virgilio, R., Giunchiglia, F., Tanca, L. (eds.): Semantic Web Information Management—A Model-Based Perspective. Springer, Berlin (2010)
2. De Virgilio, R., Guerra, F., Velegrakis, Y. (eds.): Semantic Search Over the Web. Springer, Berlin, Heidelberg (2012)
3. De Virgilio, R., Orsi, G., Tanca, L., Torlone, R.: Nyaya: A system supporting the uniform management of large sets of semantic data. In: ICD., pp. 1309–1312. (2012)
4. Bröcheler, M., Pugliese, A., Subrahmanian, V.S.: Dogma: A disk-oriented graph matching algorithm for rdf databases. In: ISWC, pp. 97–113. (2009)
5. Fan, W., Li, J., Ma, S., Tang, N., Wu, Y., Wu, Y.: Graph pattern matching: from intractable to polynomial time. Proc. VLDB Endow. **3**(1), 264–275 (2010)
6. Zhang, S., Yang, J., Jin, W.: Sapper: subgraph indexing and approximate matching in large graphs. Proc. VLDB Endow. **3**(1), 1185–1194 (2010)
7. Wood, P.T.: Query languages for graph databases. SIGMOD Rec. **41**(1), 50–60 (2012)
8. Gallagher, B.: Matching structure and semantics : A survey on graph-based pattern matching. In: Artificial Intelligence, pp. 45–53. (2006)
9. Zhang, S., Li, S., Yang, J.: Gaddi: distance index based subgraph matching in biological networks. In: EDBT, pp. 192–203. (2009)
10. Khan, A., Li, N., Yan, X., Guan, Z., Chakraborty, S., Tao, S.: Neighborhood based fast graph search in large networks. In: SIGMOD, pp. 901–912. (2011)
11. Iordanov, B.: Hypergraphdb: A generalized graph database. In: WAIM Workshops, pp. 25–36. (2010)
12. Fellbaum, C. (ed.): WordNet An Electronic Lexical Database. The MIT Press, Cambridge (1998)
13. Hassanzadeh, O., Consens, M.P.: Linked movie data base (triplification challenge report). In: I-SEMANTICS, pp. 194–196 (2008)
14. Bizer, C., Schultz, A.: The Berlin sparql benchmark. Int. J. Semant. Web. Inf. Syst. **5**(2), 1–24 (2009)
15. Guo, Y., Pan, Z., Heflin, J.: Lubm: a benchmark for owl knowledge base systems. J. Web. Semant. **3**(2–3), 158–182 (2005)

16. Ma, L., Yang, Y., Qiu, Z., Xie, G.T., Pan, Y., Liu, S.: Towards a complete owl ontology benchmark. In: ESWC, pp. 125–139. (2006)
17. 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)
18. Zou, L., Chen, L., Özsu, M.T.: Distance-join: pattern match query in a large graph database. Proc. VLDB Endow. **2**(1), 886–897 (2009)
19. Fan, W., Bohannon, P.: Information preserving xml schema embedding. ACM Trans. Database Syst. **33**(1) (2008)
20. 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 Conference, pp. 405–416 (2009)
21. Neumann, T., Weikum, G.: x-rdf-3x: fast querying, high update rates, and consistency for rdf databases. Proc. VLDB Endow. **3**(1), 256–263 (2010)
22. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: SIGMOD, pp. 335–346. (2004)
23. Zhang, S., Hu, M., Yang, J.: Treepi: A novel graph indexing method. In: ICDE, pp. 966–975. (2007)
24. Cheng, J., Ke, Y., Ng, W., Lu, A.: Fg-index: towards verification-free query processing on graph databases. In: SIGMOD, pp. 857–872. (2007)
25. Tian, Y., Patel, J.M.: Tale: A tool for approximate large graph matching. In: ICDE, pp. 963–972. (2008)
26. Zeng, Z., Tung, A.K.H., Wang, J., Feng, J., Zhou, L.: Comparing stars: on approximating graph edit distance. Proc. VLDB Endow. **2**(1), 25–36 (2009)
27. Jin, R., Xiang, Y., Ruan, N., Fuhry, D.: 3-hop: a high-compression indexing scheme for reachability query. In: SIGMOD, pp. 813–826. (2009)
28. Poulovassilis, A., Wood, P.T.: Combining approximation and relaxation in semantic web path queries. In: ISWC, pp. 631–646. (2010)
29. Chan, E.P.F., Lim, H.: Optimization and evaluation of shortest path queries. VLDB J. **16**(3), 343–369 (2007)
30. Hu, W., Jian, N., Qu, Y., Wang, Y.: Gmo: A graph matching for ontologies. In: Integrating Ontologies. (2005)