Answering top-K query combined keywords and structural queries on RDF graphs

Peng Peng\(^a,b\), Lei Zou\(^b,c\), Zheng Qin\(^a\)

\(^a\) Big Data Provincial Key Laboratory, Hunan University, Changsha, China
\(^b\) Institute of Computer Science and Technology, Peking University, Beijing, China
\(^c\) Beijing Institute of Big Data Research, Beijing, China

**Abstract**

Although SPARQL has been the predominant query language over RDF (Resource Description Framework) graphs, some query intentions cannot be captured well using only SPARQL syntax. On the other hand, keyword search enjoys widespread usage because of its intuitive way of specifying information needs, but suffers from the problem of low precision. To maximize the advantages of both SPARQL and keyword search, we introduce a novel paradigm that combines them and propose a hybrid query (called a SPARQL-Keyword (SK) query) that integrates SPARQL and keyword search. To answer SK queries efficiently, we propose a novel integrated query algorithm based on a structural index. We also present a distance-based optimization technique to further improve the efficiency of SK queries evaluation. We test our method in three large real RDF graphs and the experiments demonstrate both the effectiveness and efficiency of our method.

© 2017 Elsevier Ltd. All rights reserved.

1. Introduction

As more and more knowledge bases become available, the question of how end users can access this body of knowledge becomes crucially important. As the de facto standard of a knowledge base, an RDF (Resource Description Framework) repository is a collection of triples, denoted as \( \langle \text{subject}, \text{predicate}, \text{object} \rangle \). An RDF repository can be represented as a graph, where subjects and objects are vertices connected by labeled edges (i.e., predicates). Fig. 1 shows an example RDF dataset and the corresponding RDF graph, which is a part of the well-known knowledge base Yago [1]. All subjects and objects correspond to vertices and the predicates correspond to edge labels. The numbers beside the vertices are IDs, and they are introduced for the ease of presentation.

As we know, the SPARQL query language is a standard way to access RDF data and is based on the subgraph (homomorphism) match semantic [2]. Fig. 3(a) shows an example of a SPARQL query, and its corresponding query graph is shown in Fig. 2. The query semantics of the example SPARQL query is to “find all actors starring in the film Philadelphia”. To use SPARQL, users should have full knowledge of the whole RDF schema. For example, users should know that predicate “actedIn” means “starring in” and Philadelphia’s URI is “Philadelphia(film)”. In real applications, it may not be practical to have full knowledge about the whole schema; thus, it may not be possible to specify an exact query criterion. The following example illustrates these challenges.

**Example 1.** Find all actors starring in the film Philadelphia who are related to the “Academy Award” and “Golden Globe Award”. Assume that we do not know the exact URIs corresponding to “Academy Award” and “Golden Globe Award”. Furthermore, there is no precise predicate corresponding to “related to”.

There are two issues in this example. First, because we do not know the URIs of “Academy Award” and “Golden Globe Award”, we should provide a keyword search paradigm that maps the keywords to the corresponding entities or classes in the RDF graphs. Existing SPARQL syntax only supports regular expressions, as shown in Fig. 3. More typographic or linguistic distances, such as string edit distance [3] and Google similarity distance [4], are desirable.

The second issue is that there is no precise predicate corresponding to “related to”. One possible solution is to use an "unknown" predicate (i.e., a variable at the predicate position); however, this predicate only finds one-hop relations. Fig. 3(b) shows a SPARQL query with an unknown predicate and the regular expression FILTER. It fails to find multiple-hop relations, which may also be informative to users; for example, Antonio Banderas, an actor starring in Philadelphia, whose wife won a “Golden Globe Award”.

\(^a\) Corresponding author.

E-mail addresses: hnu16pp@hnu.edu.cn (P. Peng), zoulei@pku.edu.cn (L. Zou), zqin@hnu.edu.cn (Z. Qin).

http://dx.doi.org/10.1016/j.is.2017.03.002
0306-4379/© 2017 Elsevier Ltd. All rights reserved.
In fact, users’ query intentions cannot be well modeled using a single query type in many real-life applications. Hence, a hybrid search capability is desired. In this paper, we propose an integrated query formulation (called a SPARQL-KeyWord (SK) query) and a solution framework by combining the advantages of SPARQL and keyword search. Generally, the results of an SK query are the k SPARQL matches that are closest to all keywords in RDF graph G, where k is a parameter given by the user. The formal definition of an SK query is given in Definition 2.2.

Let us recall Example 1 again. We issue the following SK query \( Q, q \). The SPARQL query graph Q is given in Fig. 2, where the keywords are \( q = \{ \text{Academy Award}, \text{Golden Globe Award} \} \). Fig. 4 shows three different results. First, there are three different subgraph matches for query Q, i.e., \( M_1, M_2 \) and \( M_3 \). Then, the keywords are matched in different literal vertices, i.e., 001, 015 and 024. The distance between a subgraph match \( M \) and a keyword in \( q \) is the shortest distance between \( M \) and a vertex containing the keyword. We find that \( M_1 \) is the closest to the two keywords. It says “Joanne Woodward, starring in Philadelphia, won both an Academy Award and a Golden Globe Award”. Obviously, this is an informative answer to the query in Example 1.

In the above analysis, we assume that the relation strength depends on the path length, i.e., the number of hops. Actually, different predicates should have different weights for relation strength evaluation. For example, there are two two-hop paths from 021 (AntonioBanderas) to 017 (JoanneWoodward). The first one is through 020 (Philadelphia(film)), while the second is through 008 (Actor). It is obvious that two people appearing in the same film is more meaningful than the fact that both of them are actors, so the two-hop path through 020 has more relation

This is also a possible interesting result to users, but the relation between Antonio Banderas and “Golden Globe Award” is a two-hop relation.

In contrast, keyword search [5–10] on graphs provides an intuitive way of specifying information needs. For example, we may input two keywords, “Joanne Woodward” and “Golden Globe Award”, to discover unbounded relations (i.e., the paths in the RDF graphs) between them. However, keyword search may then return a larger number of non-informative search answers to users.
strength than the two-hop path through 008. Therefore, following
the intuition of TF-IDF for measuring the word importance in a
TM corpus, we propose predicate salience (see Section 2) to evaluate
relation strengths.

Another challenge of this problem is the search efficiency. A
naïve exhaustive-computing strategy works as follows: we first find
all subgraph matches of Q (in RDF graph G) using existing tech-
niques. Then, we compute the shortest path distances between
the subgraph matches and the vertices containing keywords on
the fly. Finally, the matches with the shortest distances to the key-
words are returned as answers. Obviously, this is an inefficient so-
lution. Given a SPARQL query Q, there may exist some matches of
Q that are far from the keywords in the RDF graphs. These matches
cannot contribute to the final results. Therefore, it is unnecessary
to identify all subgraph matches in RDF graph. Instead of the ex-
haustive computing, we only find matches for SPARQL query Q pro-
gressively and design a lower bound that stops the search process
as early as possible. Moreover, we propose a star index to enable
structural pruning. To further improve the efficiency of distances
between these subgraph matches and the vertices containing key-
words, we propose a distance-based optimization to speed up the
shortest path distance computation. We select some pivots and
materialize the shortest path trees rooted at these pivots. During
the distance computation, if the traversal meets a pivot p, we util-
ize the shortest path tree rooted at p to reduce the search space.

In summary, we make the following contributions in this paper.

1. We propose a new query paradigm over RDF data combining
keywords and SPARQL (called an SK query), and design a novel
solution for this problem.

2. We design an index and a distance-based optimization to speed
up SK query processing. We propose a frequent star pattern-
base index and materialize some shortest path trees to reduce
the search space and improve query performance.

3. We evaluate the effectiveness and efficiency of our method in
real large RDF graphs and conclude that our methods are much
better than comparative models with respect to both effective-
ness (in terms of NDCG@k) and query response time.

The remainder of this paper is organized as follows: Section 2 defines
the preliminary concepts. Section 3 gives an overview of our approach. We introduce a structural index to
efficiently find the candidates of variables in SPARQL queries in
Section 4. We discuss the manner in which the results of SK
queries are computed in Section 5. A distance-based optimization
technique is proposed in Section 6. Experimental results are pre-
sented in Section 7. Related work is reviewed and the conclusions
are drawn in Sections 8 and 9, respectively.

2. Background

In this section, we introduce the fundamental definitions used in
this paper.

2.1. Preliminaries

An RDF dataset consists of a number of triples, which corre-
sponds to an RDF graph. The SK query is to find the k SPARQL
matches that are the top-k nearest with regard to all keywords.

Fig. 4. SK query results.
Definition 2.1. An RDF data graph $G$ is denoted as $(V(G), E(G), L)$, where (1) $V(G) = V_1 \cup V_2 \cup V_3$ is the set of vertices in RDF graph $G$; $V_1$, $V_2$, and $V_3$ denote literal, entity and class vertices, respectively; (2) $E(G)$ is the set of edges in $G$; and (3) $L$ is a finite set of edge labels, i.e. predicates.

Definition 2.2. An SK (SPARQL & Keyword) query is a pair $(Q, q)$, where $Q$ is a SPARQL query graph, and $q$ is a set of keywords $\{w_1, w_2, \ldots, w_n\}$.

Given an SK query $(Q, q)$ in data graph $G$ is a pair $(M, \{v_1, v_2, \ldots, v_n\})$, where $M$ is a subgraph match of $Q$ in $G$ and $v_i (i = 1, \ldots, n)$ is a literal vertex in $G$ containing keyword $w_i$.

Given an SK query $(Q, q = \{w_1, \ldots, w_n\})$, the result of a cost $r = \langle M, \{v_1, v_2, \ldots, v_n\} \rangle$ contains two parts. The first part is the content cost and the second part is the structure cost.

Definition 2.3. Given a result $r = \langle M, \{v_1, v_2, \ldots, v_n\} \rangle$, the cost of $r$ is defined as follows:

$$\text{Cost}(r) = \text{Cost}_{\text{content}}(r) + \text{Cost}_{\text{structure}}(r),$$

where $\text{Cost}_{\text{content}}(r)$ is the content cost of $r$ (defined in Definition 2.4) and $\text{Cost}_{\text{structure}}(r)$ is the structure cost of $r$ (defined in Definition 2.5).

Definition 2.4. Given a result $r = \langle M, \{v_1, v_2, \ldots, v_n\} \rangle$, the content cost of $r = \langle M, \{v_1, v_2, \ldots, v_n\} \rangle$ is defined as follows:

$$\text{Cost}_{\text{content}}(r) = \sum_{i=1}^{n} C(v_i, w_i),$$

where $C(v_i, w_i)$ is the matching cost between $v_i$ and keyword $w_i$.

Any typographic or linguistic distances such as string edit distance [11] or Google similarity distance [4] can be used to measure $C(v_i, w_i)$.

In applications, users are more interested in some variables (in a SPARQL query $Q$) than the constants in $Q$. Let us recall Example 1. The distance between the keywords and matching vertices with regard to variable “?a” is more interesting to measure with the relationship strength. Therefore, to evaluate the structure cost (in Definition 2.5), we only consider the matching vertices with regard to the variables in SPARQL query $Q$.

Definition 2.5. Given a result $\langle M, \{v_1, v_2, \ldots, v_n\} \rangle$ for an SK query $(Q, q)$, the distance between match $M$ and vertex $v_i (i = 1, \ldots, n)$ is defined as follows.

$$d(M, v_i) = \text{MIN}_{\psi \in \psi_D} \{d(\psi, v_i)\}$$

where $v$ is a matching vertex in $M$ with regard to a variable in SPARQL query $Q$ and $d(\psi, v_i)$ is the shortest path distance between $v$ and $v_i$ in RDF graph $G$.

The structure cost of a result $r = \langle M, \{v_1, v_2, \ldots, v_n\} \rangle$ is then defined as follows.

$$\text{Cost}_{\text{structure}}(r) = \sum_{i=1}^{n} d(M, v_i)$$

(Problem Definition) Given an SK query $(Q, q)$ and parameter $k$, our problem is to find the $k$ results (Definition 2.2) that have the $k$-smallest costs.

2.2. Predicate salience

In this paper, we use the shortest path distance to evaluate the relation strength. However, the naive definition of the shortest path distance suffers from a critical problem: all predicates, i.e., edge labels, are considered equally important when it is used to measure the relationship strength between entities. In fact, some predicates have little or no discriminating power when determining relevance. For example, predicates like “type” and “label” are so common that each entity is incident to a class vertex through an edge of predicate “type”. This tends to incorrectly emphasize paths that contain these common predicates more frequently, without giving enough weight to the paths with more meaningful predicates (like “actedin” and “isMarriedTo”). Predicates such as “type” and “label” are not good predicates for distinguishing relevant and non-relevant vertices, unlike the less common predicates “actedin” and “isMarriedTo”.

Hence, we should introduce a mechanism for attenuating the effect of predicates that occur too frequently in the RDF graph to be meaningful for relevance determination. Learning from the concept of document frequency, we first determine the set of vertices in the RDF graph incident to a predicate $p$, which is denoted as $V(p)$. We then divide the size of $V(p)$ by the total number of vertices. We name this measure as the predicate salience of predicate $p$ and give its formal definition as follows:

$$ps(p) = \frac{|V(p)|}{|V(G)|}.$$ 

Thus the predicate salience of a rare predicate is low, whereas the predicate salience of a frequent predicate is likely to be high, which means that rare predicates have less cost than frequent predicates.

Let us consider the RDF graph in Fig. 1. The predicate salience values of all predicates are given in Table 1. As shown in Table 1, predicate “actedin” is more important than “type” when measuring the relation strength; the former’s predicate salience is 0.296 and the latter’s is 0.593.

3. Overview

In this section, we give an overview of the different steps involved in our SK query process, which is depicted in Fig. 5. In this paper, we are concerned with the challenge of efficiently finding the results of SK queries. We propose an approach in which the best results of the SK query are computed using graph exploration. We detail the different steps of the approach below.

**Keyword Mapping.** In the offline phase, we create an inverted index storing a map from keywords to its locations in the RDF graph. In the online phase, we map keywords to vertices based on the inverted index.

For scoring keyword vertices, a widely used metric that is computed on-the-fly for a given query is IR-style TF/IDF cost. Many cost functions have been proposed in the literature, and we select one of them to assign the cost to each vertex containing keywords. Note that we need to normalize the cost of keyword matching vertices before the distance computations.

In this paper, our primary focus is how to find the matches of a SPARQL query and their relations to keywords. We use an existing IR engine to analyze given keywords, perform an imprecise matching, and return a list of graph elements having labels that are syntactically or semantically similar. We do not delve into the specifics of keyword mapping.
**Candidate Generation.** When we find a vertex reachable to elements of all keywords, we need to run a subgraph homomorphism to check whether there exist some subgraph matches (of Q) containing v. As we know, a subgraph homomorphism is not efficient because of its high complexity [12]. To speed up query processing, we propose a filter-and-refine strategy to reduce the number of subgraph homomorphism operations. The basic idea is to filter out some vertices that are not in any subgraph match of Q. We call them dummy vertices. If the search meets a dummy vertex, we do not perform the subgraph homomorphism algorithm.

In this paper, we propose a frequent star pattern-based structural index. Based on this index, we can locate a candidate list in the RDF graph of each variable in the SPARQL query. A vertex in at least one variable candidate list is not dummy. We detail how to build the structural index in Section 4.1 and how to use the index to reduce the candidates of all variables in Section 4.2.

**Top-k Results Computation.** Based on the keyword vertices and variables’ candidates, we propose a solution based on graph exploration to compute the top-k results of SK queries. Our approach starts graph exploration from all keyword vertices and explores their neighboring vertices recursively until the distances between a vertex and the keyword vertices have been computed. When the distances between a vertex and the vertices of all keywords have been computed, we check whether this vertex is a dummy vertex. If so, there exists no match of Q that can contain it. Hence, we can skip it. Otherwise, we start our SPARQL matching algorithm (Algorithm 2) from the vertex to generate all matches containing it. The exploration terminates when the top-k results have been computed. We propose some early stop strategies so that the top-k computation will reach early termination after obtaining the top-k results, instead of searching the data graph for all results. Furthermore, we materialize some shortest path trees as the distance-based index to speed up the top-k results computation.

We discuss the details of top-k results computation and our distance-based optimization technique in Sections 5 and 6.

### 4. Candidate generation based on the structural index

In this section, we first introduce a structural index based on a certain family of patterns in Section 4.1. We then discuss how to generate the candidate lists of variables based on our structural index in Section 4.2.

#### 4.1. Structural index

In this section, we propose a frequent star pattern-based index. We mine some frequent star patterns in G. For each frequent star

![Fig. 5. Overview of our approach.](image)

<table>
<thead>
<tr>
<th>Table 2 Examples of predicate sequences.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vertex</strong></td>
<td><strong>Predicate Sequence</strong></td>
</tr>
<tr>
<td>Philadelphia (film)</td>
<td>(actedIn, actedIn, actedIn, label, type)</td>
</tr>
<tr>
<td>Joanna Woodward</td>
<td>(actedIn, actedIn, label, type, wonPrize)</td>
</tr>
<tr>
<td>Antonio Banderas</td>
<td>(actedIn, label, type, wonPrize, wonPrize)</td>
</tr>
</tbody>
</table>

S, we build an inverted list \( L(S) \) that includes all vertices (in RDF graph \( G \)) contained by at least one match of \( S \). A reason for selecting the stars as index elements is that SPARQL queries tend to contain star-shaped subqueries for combining several attribute-like properties of the same entity [13].

We propose a sequential pattern mining-based method to find frequent patterns in RDF graphs. For each entity vertex in an RDF graph, we sort all its adjacent edges in lexicographic order of edge labels (i.e., properties). These sorted edges can form a sequence. For example, vertex “Philadelphia(film)” has five adjacent edges, which are (actedIn, actedIn, actedIn, name, type). Table 2 shows a sequence database, where each sequence is formed by the adjacent edges of one entity vertex. We employ existing sequential pattern mining algorithms, such as PrefixSpan [14] to find frequent sequential patterns, where each sequential pattern corresponds to a star pattern in RDF graphs. For example, assume that minimal support count \( s = 2 \), and (actedIn, type) and (actedIn, type, wonPrize) are two frequent sequential patterns. It is easy to see that a sequential pattern always corresponds to a star pattern, as shown in Fig. 6. For ease of presentation, we use the terms “sequential patterns” and “star patterns” interchangeably in the following discussion.

For each frequent star pattern \( S \), we maintain an inverted list \( L(S) = \{v \mid S \text{ occurs in } v \text{'s adjacent edge sequence} \} \). Obviously, if we use all frequent stars as the index elements, the space cost is very large. Thus, inspired by gndex [15], we also define a **discriminative ratio** for the star pattern selection.

**Definition 4.1.** Given a star \( S \), its discriminative ratio is defined as follows:

\[
\gamma(S) = \frac{|L(S)|}{|\cap_{S < S'} L(S')|}
\]
where \( S' \subseteq S \) denotes that \( S' \) is a part of \( S \).

Obviously, \( \gamma(S) \leq 1 \). Further, \( \gamma(S) = 1 \) means that \( L(S) \) can be obtained by the intersection of all \( L(S') \), where \( S' \subseteq S \). In this case, if all \( S' \) are index elements, it is not necessary to keep \( S \) as the index element, as \( S \) cannot provide more pruning power. In practice, we set a threshold \( \gamma_{\text{max}} \), and we only choose stars \( S \) such that \( \gamma(S) \leq \gamma_{\text{max}} \). Note that to ensure the completeness of the indexing, we always choose the (absolute) support to be 1 for size-1 stars (stars with only one edge). This method guarantees no-false-positives, as all vertices (in \( G \)) are indexed in at least one inverted list.

**Theorem 4.1.** Let \( F \) denote all selected index elements (i.e., frequent star patterns). Given a SPARQL query \( Q \), a vertex \( v \) in graph \( G \) can be pruned (there exists no subgraph match of \( Q \) containing \( v \)) if the following equation holds.

\[
v \notin \bigcup_{S \in F \setminus S_Q} L(S).
\]

where \( S \in F \) means that \( S \) is a selected star pattern and \( S \in Q \) is a star pattern included in \( Q \).

**Proof.** If \( v \notin \bigcup_{S \in F \setminus S_Q} L(S) \), it means that the structure around \( v \) does not contain any substructure of \( Q \). Hence, \( v \) must be unable to be found in a match of \( Q \). \( \square \)

### 4.2. Candidate generation

Given an SK query, we first tag the vertices that can be pruned by **Theorem 4.1**. For each variable in SPARQL, we locate its candidates in an RDF graph. Each variable can map to a predicate sequence according to the SPARQL statement. For example, variable “\( qa \)” of the SPARQL query in Fig. 2 has the predicate sequence \((\text{act}edIn, \text{type})\). Then, for each variable \( x \), we look up our structural index and find the maximum pattern contained by \( x \)'s predicate sequence. We load the vertex list of the maximum pattern as \( x \)'s candidates. A vertex in at least one vertex list of variables is not a dummy. We define the pruned vertices as dummy vertices, as follows.

**Definition 4.2. Dummy Vertex.** Given a SPARQL query \( Q \), a vertex \( v \) in graph \( G \) is called a dummy vertex if the following equation holds.

\[
v \notin \bigcup_{S \in F} L(S).
\]

where \( F \) denotes all selected frequent star patterns, \( S \in F \) means that \( S \) is a selected star pattern and \( S \in Q \) is a star pattern included in \( Q \).

When the search process meets a fully seen vertex \( v \), if \( v \) is not a dummy vertex, we perform a subgraph homomorphism algorithm to find the subgraph match of the SPARQL query \( Q \) containing \( v \). Otherwise, we do not perform a subgraph homomorphism algorithm beginning with \( v \).

### 5. Top-k results computation

In this section, we introduce our approach for SK queries, which is based on the backward search strategy [5]. Our algorithm for searching for top-k results of SK queries is shown in **Algorithm 1**. This algorithm consists of three parts: (1) graph exploration to find vertices connecting the keyword vertices, (2) generation of SPARQL matches from the vertices connecting the keyword vertices, and (3) top-k computation. In the following, we elaborate on these three tasks.

#### 5.1. Graph exploration

Given the keyword vertices, the objective of the exploration is to find vertices in the graph that connect with these keyword vertices and compute their distances. Let \( V_l \) denote all literal vertices (in RDF graph \( G \)) containing keyword \( w_l \).

**Definition 5.1. Distance between a Vertex and Keyword.** Given a vertex \( v \) in RDF graph \( G \) and a keyword \( w_l \), the distance between \( v \) and keyword \( w_l \) (denoted as \( d(v, w_l) \)) is the minimum distance between \( v \) and a vertex in \( V_l \), where \( V_l \) includes all literal vertices containing keyword \( w_l \) in \( G \).

For graph exploration, we maintain a priority queue \( PQ \) for each keyword \( w_l \). Each element in \( PQ \) is represented as \((v, p, |p|)\), where \( v \) is a vertex ID, \( p \) is a path between \( v \) and a vertex in \( V_l \) and \(|p|\) denotes the path distance. All elements in \( PQ \) are sorted in non-descending order of \(|p|\). Each keyword \( w_l \) is also associated with a result set \( R_{w_l} \). To keep track of information related to each vertex \( v \), we associate \( v \) with a vector \( d[v] \). If a vertex \( v \) is in \( R_{w_l} \), the shortest path distance is known. In this case, we set \( d[v][i] = d(v, w_l) \); otherwise, we set \( d[v][i] = \text{null} \).

Initially, the exploration starts with a set of vertices containing keywords. For each vertex \( v \) containing keyword \( w_l \), an element \((v, 0, 0)\) is created and placed into the queue \( PQ \) (Line 3 in **Algorithm 1**). During the search, at each step, we pick a queue \( PQ \) \((i = 1, \ldots , n)\) to expand in a round-robin manner (Line 5 in **Algorithm 1**). We assume that we pop the queue head \((v, p, |p|)\) from \( PQ \). When a queue head \((v, p, |p|)\) is popped from queue \( PQ \), we insert it into result set \( R_{w_l} \) and set \( d[v][|p|] = |p| \) (Line 6 in **Algorithm 1**). We prove that the following theorem holds.

**Algorithm 1:** Search for top-k results of SK queries.

**Input:** RDF data graph \( G \), SK query \((Q, q)\), \([V_1, \ldots , V_s]\) where \( V_i \) is the set of vertices containing keyword \( w_i \), priority queues \([PQ_1, \ldots , PQ_n]\).

**Output:** Top-k results \( R \) of \((Q, q)\).

1. for each vertices set \( V_l \) do
   2. for each vertex \( v \) in \( V_l \) do
     3. Insert \((v, 0, 0)\) into \( PQ \);
   4. while not all queues are empty and \( \theta \geq \delta \) do
     5. for \( i = 1, \ldots, n \) do
       6. Pop the head of \( PQ_i \) \((v, p, |p|)\), set \( d[v][i] = |p| \) and insert it into \( R_{w_{PQ_i}} \);
       7. for each adjacent edge \( (v, p, \overrightarrow{w}) \) to \( v \) do
         8. if \( p \cup \overrightarrow{w} \) is not a simple path then
           9. Continue;
         10. if there exists another element \((v', p', |p'|)\) in \( PQ_i \) then
           11. if \(|p'| > |p| + ps(\overrightarrow{w})\) then
             12. Replace \((v', p', |p'|)\) with \((v', p \cup \overrightarrow{w}, |p| + ps(\overrightarrow{w}))\) in \( PQ_i \);
           13. else
             14. Insert \((v', p \cup \overrightarrow{w}, |p| + ps(\overrightarrow{w}))\) in \( PQ_i \);
         15. if \( v \) is a fully seen vertex then
           16. Call Algorithm 2 to find all matches containing \( v \);
           17. for each match \( M \) that are found do
             18. if all vertices in \( M \) are fully seen vertices then
               19. Use \( M \) to update \( R \) and the upper bound \( \delta \) of top-k results;
             20. Update the cost of all partially seen matches and \( \delta \);
             21. Update the lower bound cost \( \theta \) of all remaining unseen vertices;
       22. Return \( R \).
Theorem 5.1. When a queue head \((v, p, [p])\) is popped from queue \(PQ_i\), the following equation holds.
\[
d(v, w_i) = d[v][i] = |p|
\]

Proof. Given a vertex \(v\) before \((v, p, [p])\) is popped from \(PQ_i\) and a path \(p\) between \(v\) and vertices containing \(w_i\), it is obvious that \(|p| \geq d(v, w_i)\).

We wish to show that in each iteration, \(d(v, w_i) = d[v][i] = |p|\) for the element \((v, p, [p])\) popped from \(PQ_i\). We prove this by contradiction. Assume that \(v\) is the first vertex for which \(d[v][i] = |p| \neq d(w_i, v)\) when \((v, p, [p])\) is popped from \(PQ_i\). We focus our attention on the situation at the beginning of the iteration in which \((v, p, [p])\) is popped from \(PQ_i\) and then derive the contradiction that \(d[v][i] = |p| = d(v, w_i)\) by examining the shortest path from \(v\) to the vertices containing \(w_i\). We must have \(v \in V_i\) because all vertices in \(V_i\) are the first vertices added to set \(S_i\) and \(d[v][i] = 0\) at that time.

Because \(v \in V_i\), we also have that \(S_i \neq \emptyset\) just before \((v, p, [p])\) is popped from \(PQ_i\). There must be some paths from vertices containing \(w_i\) to \(v\) for otherwise \(d[v][i] = \infty\) by the no-path property, which would violate our assumption that \(d[v][i] \neq d(w_i, v)\). Because there is at least one path, there is a shortest path \(p'\) between \(v\) and vertices in \(V_i\). Prior to the pop of \((v, p, [p])\) from \(PQ_i\), path \(p'\) connects a vertex in \(S_i\), namely some vertices in \(V_i\), to a vertex in \(V(G) - S_i\), namely \(v\). Let us consider the first vertex \(v'\) along \(p'\) such that \(v' \in V(G) - S_i\), and let \(v'' \in S_i\) be the predecessor of \(v'\).

We claim that \(d[v'][i] = d(w_i, v')\) when the element of \(v'\) is popped from \(PQ_i\). To prove this claim, observe that \(v'' \in S_i\). Then, because \(v\) is chosen as the first vertex for which \(d[v][i] \neq d(w_i, v)\) when \((v, p, [p])\) is popped from \(PQ_i\), we have \(d[v'][i] = d(w_i, v')\) when \(v'' \in S_i\) is added to \(S_i\). Edge \(v''v\) is relaxed at that time (Lines 7–17 in Algorithm 1), so the claim follows from the convergence property.

We can now obtain a contradiction to prove that \(d[v'][i] = d(v, w_i)\). Because \(v''\) occurs before \(v\) on the shortest path from vertices in \(V_i\) to \(v\) and all edge weights are nonnegative, we have \(d[v'][i] \leq d(v, w_i)\), and thus \(d[v', w_i] = d[v'][i] \leq d(v, w_i) \leq d[v][i]\).

However, because both vertices \(v\) and \(v''\) are in \(V(G) - S_i\) when \(v''\) is popped before \(v\), we have \(d[v', w_i] \leq d(v, w_i)\). Thus, \(d[v', w_i] = d[v'][i] = d(v, w_i)\), which contradicts our choice of \(v\). We conclude that \(d[v][i] = d(w_i, v)\) when \((v, p, [p])\) is popped from \(PQ_i\), and that this equality is maintained at all times thereafter. \(\Box\)

When a queue head \((v, p, [p])\) is popped from queue \(PQ_i\), it means that we have computed the distance between \(v\) and keyword \(w_i\). We also say that \(v\) is seen by keyword \(w_i\).

Definition 5.2. Seen by a Keyword. When queue head \((v, p, [p])\) is popped from queue \(PQ_i\), the distance between \(v\) and keyword \(w_i\) has been computed. We then say that vertex \(v\) is seen by keyword \(w_i\).

Algorithm 2: SPARQL matching algorithm.

Input: A candidate vertex \(v\) corresponding to \(u\) in SPARQL query \(Q\), and a state stack \(S\).

Output: The match set \(MS\) of \(Q\) containing \(v\).

1. Initialize a state \(s\) with \(v\);
2. Push \(s\) into \(S\);
3. while \(S \neq \emptyset\) do
4. Pop the first state \(s\) in \(S\);
5. if all edges of \(Q\) have been matched in \(s\) then
6. Insert \(s\) to \(MS\);
7. for each unmatched edge \(u'v''\) that \(u'\) has been matched to \(v''\) do
8. if \(u''\) has been matched to \(v''\) then
9. if \(\mathcal{E}(G)\) then
10. Push \(s\) into \(S\);
11. else
12. for each neighbor \(v''\) of \(v''\) do
13. if \(v''\) is a dummy or fully seen vertex then
14. Continue;
15. if \(\mathcal{E}(G)\) can match \(u''v''\) then
16. Initialize a new state \(s'\) and \(s' = s\);
17. Match \(u''\) with \(v''\);
18. Push \(s\) into \(S\);
19. Return \(MS\).
v's neighbor v' corresponding to u' in Q, where u' is one of u's neighbors and edge uv satisfies query edge u'v'. The search will extend the state step by step. The search branch terminates when we have found a state corresponding to a match or we cannot continue. In this case, the algorithm backtracks to some other states and tries other search branches.

As shown in Algorithm 2, we find all matches containing some fully seen vertex v only if v is not a dummy vertex (Lines 15-16 in Algorithm 2). This is because there exists no subgraph match containing a dummy vertex. When we finish Algorithm 2 from v, we say that v is has been searched. The term “searched” indicates that all matches containing v have been found, if any. When we search the RDF graph beginning with a fully seen vertex, if the search meets another fully seen vertex v', it can skip v' (Lines 15-16 in Algorithm 2). This is because the matches containing v' have been found before.

Example 2. We assume that the current popped fully seen vertex is 020 (Philadelphia(films)) and vertex 017 (JoanneWoodward) is another a fully seen vertex. As shown in Fig. 8, we explore the RDF graph from 020 to 017. However, vertex 017 is a fully seen vertex, so all SPARQL matches containing “017” have been found already. Thus, we can terminate the corresponding search branches in Fig. 8.

5.3. Top-k computation

The native solution for computing the top-k results of a SK query is to run the backward search algorithm until all vertices (in RDF graph G) have been fully seen by the keywords. Then, according to the results’ cost, we can find the top-k results. Obviously, this is an inefficient solution especially when G is very large. In this subsection, we design an early-stop strategy.

Let us consider a snapshot of an iteration step in Algorithm 1. All subgraph matches of SPARQL query Q can be divided into three categories: fully seen matches, partially seen matches, and unseen matches.

![Fig. 7. Example state.](image)

![Fig. 9. Fully seen match, partially seen match and unseen match during the top-k results computation.](image)

Definition 5.5. Fully Seen Match, Partially Seen Match and Unseen Match. Given a subgraph match M of SPARQL query Q, if all vertices in M are fully seen vertices, M is called a fully seen match; if M is not a fully seen match and M contains at least one fully seen vertex, it is called a partially seen match. If a match M does not contain any fully seen vertex, it is called an unseen match.

Fig. 9 demonstrates a visual representation of three kinds of matches. The shaded area covered by the dash line circle denotes all fully seen vertices in RDF graph. As the iteration steps (in Algorithm 1) increases, the shaded area expands gradually until it covers the whole RDF graph. The early-stop strategy is to stop the expansion as early as possible, but we can guarantee that we have found the top-k results for an SK query.

The basic idea of our early-stop strategy is as follows. We only compute the cost of fully seen matches. We then use the fully seen matches to find a threshold δ, which is the k-th smallest cost so far. If there are less than k fully seen matches so far, δ is ∞. We compute the lower bounds θ1 and θ2 for partially seen matches and unseen matches, respectively. The algorithm can stop early if and only if δ < θ1 and δ < θ2. Otherwise, the algorithm continues to the next iteration.

**Fully Seen Match.** For a fully seen match, we compute its match cost according to Definition 2.5. If we have found more than k fully seen matches, we maintain a threshold δ, which is the k-th smallest match cost.

![Fig. 8. Finding matches containing vertex 020 by pruning the search branch beginning from 017.](image)
**Partially Seen Match.** For any partially seen match, we compute the lower bound of its cost as follows.

**Theorem 5.2.** Given a partially seen match M of SPARQL query Q, v is a partially seen or an unseen vertex in the match. The following equation holds.

\[
\text{Cost}(M) = \sum_{1 \leq i \leq n} d(v, w_i) \geq \sum_{d(v[w_i]) = \text{null} \land 1 \leq i \leq n} d(v[w_i]) + \sum_{d(v[w_i]) = \text{null} \land 1 \leq i \leq n} |p_i|.
\]

where \(d(v[w_i])\) is the i-th dimension of v's vector corresponding to keyword \(w_i\), and \(|p_i|\) corresponds to the current queue head \((v, p_i, |p_i|)\) in queue PQ.

**Proof.** If \(d(v[w_i]) \neq \text{null}\), it means that we have computed \(d(v, w_i)\). If \(d(v[w_i]) = \text{null}\), v has still not been seen. Because each time we pop the head \((v, p_i, |p_i|)\) of PQ, where \(|p_i|\) is the smallest, all unseen vertices' distances to w_i are larger than \(|p_i|\).

According to Theorem 5.2, we define the lower bound of a partially seen match as follows.

**Definition 5.6.** Given a match of SPARQL query Q, the lower bound for a partially seen match M is defined as follows.

\[
\text{lb}(M) = \text{MIN}_{p \in \text{PS}}(\sum_{d(v[w_i]) = \text{null} \land 1 \leq i \leq n} d(v[w_i]) + \sum_{d(v[w_i]) = \text{null} \land 1 \leq i \leq n} |p_i|).
\]

The lower bound for all partially seen matches is defined as follows.

**Definition 5.7.** The lower bound \(\theta_1\) for all partially seen matches is as follows.

\[
\theta_1 = \text{MIN}_{M \in \text{PS}}(\text{lb}(M))
\]

where PS denotes all partially seen matches and lb(M) is defined in Definition 5.6.

As the iteration steps progress, some partially seen matches become fully seen matches. The threshold \(\delta\) and \(\theta_1\) are updated accordingly.

**Unseen Match.** Let us consider an unseen match M. There are two kinds of vertices in M, i.e., partially seen vertices and unseen vertices.

**Theorem 5.3.** For an unseen vertex v, if threshold \(\delta \neq \infty\), the following equation holds.

\[
\delta \leq \sum_{1 \leq i \leq n} d(v, w_i)
\]

**Proof.** For each keyword \(w_i\), we assume that the queue head of PQ_i is \((v, p_i, |p_i|)\). Because v is an unseen vertex, \(|p_i| \leq d(v, w_i)\) for each keyword \(w_i\). In contrast, \(\delta \) is the upper bound of the top-k results, so \(\delta \) is equal to the cost of a fully seen match M. Each vertex \(v'\) in \(M\) is fully seen vertex. Hence, \(d(v', w_i) \leq |p_i|\). We then know that \(d(v', w_i) \leq |p_i| \leq d(v, w_i)\) for each keyword \(w_i\). In conclusion, \(\delta \leq \sum_{1 \leq i \leq n} d(v, w_i)\).

According to Theorem 5.3, it is not necessary to consider unseen vertices to define the lower bound for unseen matches. Therefore, we define the lower bound for all unseen matches as follows.

**Definition 5.8.** The lower bound \(\theta_2\) for all unseen matches is as follows.

\[
\theta_2 = \text{MIN}_{p \in \text{PS}}(\sum_{d(v[w_i]) = \text{null} \land 1 \leq i \leq n} d(v[w_i]) + \sum_{d(v[w_i]) = \text{null} \land 1 \leq i \leq n} |p_i|).
\]

where PSet contains all partially seen vertices so far, \(d(v[w_i])\) is the i-th dimension of v's vector corresponding to keyword \(w_i\), and \(|p_i|\) corresponds to the current queue head \((v, p_i, |p_i|)\) in queue PQ.

**Early-stop Strategy.** At each iteration step, we check whether \(\delta \leq \theta_1 \land \delta \leq \theta_2\). If the condition holds, the algorithm can stop, as any partially seen match or unseen match cannot be in one of the top-k results.

6. **Distance-based optimization**

As discussed in Section 5, Algorithm 1 employs the backward search strategy to traverse over a large RDF graph online. Obviously, it is not efficient. To speed up the traversal, we propose a pivot-based distance index in this section. Specifically, we select some vertices as pivots. We compute the shortest path trees rooted at these pivots in the offline process. During the backward search, if the traversal meets a pivot p, we can utilize the shortest path tree rooted at p to reduce the search space.

6.1. **Pivot-based search for top-k results of SK queries**

In this section, we will discuss how we use pivots and their shortest path trees to speed up the online backward search over the RDF data graph.

First, we introduce the background of our idea. Let us recall the search from keyword “Golden Globe Award” in our running example. Here, keyword “Golden Globe Award” only maps vertex 015 ("Golden Globe Award for Best Actress"), and we assume that keyword “Golden Globe Award” maps to vertex 015 with content cost 0. The shortest path tree rooted at vertex 015 is given in Fig. 10.

**Definition 6.1.** Given a shortest path tree T rooted at vertex r (denoted as T(r)), pivot p, and vertex v, if the shortest path between r and v crosses pivot p, we say that v is covered by p in T.

For example, we assume that there are two pivots: 017 (JoanneWoodward) and 027 (Mogambo). Some parts of the shortest path tree are covered (defined in Definition 6.1) by the two pivots, as shown in Fig. 10.

**Theorem 6.1.** If v is covered by p in the shortest path tree T(r), \(d(r, v) = d(r, p) + d(p, v)\) where \(d(r, v)\) denotes the shortest path distance between r and v.

**Proof.** According to Definition 6.1, because v is covered by p, the shortest path \(\pi\) between r and v crosses pivot p. Hence, there are two parts in \(\pi: \pi_1\) from r to p and \(\pi_2\) from p to v. To prove that \(d(r, v) = d(r, p) + d(p, v)\), we only need prove that \(\pi_1\) and \(\pi_2\) are the shortest paths from r to p and from p to v, respectively.

First, we prove that \(\pi_1\) is the shortest path from r to p. If \(\pi_1\) is not the shortest path, there exists a path \(\pi'_1\) from r to p where \(|\pi'_1| < |\pi_1|\). We can then create new path \(\pi'\) of length \(|\pi'_1| + |\pi_2|\) by concatenating paths \(\pi'_1\) and \(\pi_2\). However, \(|\pi'\| = |\pi'_1| + |\pi_2| < |\pi_1| + |\pi_2| = |\pi|\). This conflicts with the fact that \(\pi\) is the shortest path between r and v. Therefore, \(\pi_1\) is the shortest path from r to p and \(d(r, p) = |\pi_1|\).

Similarly, we can prove that \(\pi_2\) is the shortest path from p to v and \(d(p, v) = |\pi_2|\). Hence, we have that \(d(r, v) = |\pi| = |\pi_1| + |\pi_2| = d(r, p) + d(p, v)\).

For example, as shown in Fig. 10, vertex 001 is covered by pivot 017 in T(015), \(d(015, 001) = d(015, 017) + d(017, 001) = 2.222\).

Motivated by the above observation, we extend our search algorithm to a pivot-based search algorithm, as shown in Algorithm 3.

We assume that keyword \(w\) maps to vertices in \(V\). Initially, we insert all vertices into result set \(S_0\), and all initial distances \(d(v[w])\) (\(\forall v \in V\)) are set to \(+\infty\) except for vertices in \(V_1\), whose distances are their content costs. The differences between the pivot-based search algorithm and the basic counterpart only happen when a queue head \(h\) popped from the queue PQ is a pivot (see Lines 7–17 in Algorithm 1).
Algorithm 3: Pivot-based search for top-k results of SK queries.

Input: RDF data graph G, SK query \((Q, q)\), \(V_1, \ldots, V_n\) where \(V_i\) is the set of vertices containing keyword \(w_i\), priority queues \(PQ_1, \ldots, PQ_n\), the set of pivot, \(PV\), and all shortest path trees of vertices in \(PV\).

Output: Top-k results \(R\) of \((Q, q)\).

1. for each vertex set \(V_i\) do
2.   for each vertex \(v\) in \(V_i\) do
3.     Insert \((v, \theta, O)\) into \(PQ_i\);
4.   while not all queues are empty and \(\theta \geq \delta\) do
5.     for \(i = 1, \ldots, n\) do
6.       Pop the head of \(PQ_i\) \((v, p, |p|)\), set \(d(v)[i] = |p|\) and insert it into \(RS_1\);
7.       if \(v \in PV\) then
8.         Update all vertices’ distance to keyword \(w_i\) based on \(v\)’s shortest path tree;
9.       else
10.      for each adjacent edge \(e \in v\) do
11.        if \(p + v\) is not a simple path then
12.           Continue;
13.        if there exists another element \((v', p', |p'|)\) in \(PQ_i\) then
14.           if \(|p'| > |p| + ps(v)\)) then
15.             Replace \((v', p', |p'|)\) with \((v, p \cup v, |p| + ps(v))\) in \(PQ_i\);
16.           else
17.             Insert \((v', p \cup v, |p| + ps(v))\) in \(PQ_i\);
18.       if \(PQ_i\) is empty then
19.         for each fully seen vertex \(v'\) in \(PQ_i\) do
20.           Call Algorithm 2 to find all matches containing \(v'\);
21.         for each match \(M\) that are found do
22.           if all vertices in \(M\) are fully seen vertices then
23.             Use \(M\) to update \(R\) and the upper bound \(\delta\) of top-k results
24.             Update the cost of all partially seen matches and \(\delta\);
25.             Update the lower bound cost \(\theta\) of all remaining un-seen vertices;
26.       Return \(R\).

When \(v\) is popped and is a pivot, it means that \(d[v][i]\), the shortest path between \(v\) and keyword \(w_i\), has been computed. We update \(d[v][i] = |p|\). In addition, we load the shortest path tree \(T(v)\). For each vertex \(v'\) in \(G\), if \(|p| + d(v, v') < d[v][i]\), we update \(d[v][i] = |p| + d(v, v')\), where \(d(v, v')\) is the distance from \(v\) to \(v'\) that can be obtained from \(T(v)\). In contrast to the basic search algorithm, it is not necessary to put all the neighbors of \(v\) into queue \(PQ_i\).

In addition, because of the pivots, the search algorithm may be terminated while some vertices have not yet been popped. For these vertices, we find all matches containing them when the priority queue is empty (see Lines 18–20 in Algorithm 1).

Example 3. Fig. 10 shows how our pivot-based search algorithm runs while evaluating keyword “Golden Globe Award” in our running example. Initially, we push all vertices onto \(RS_2\) and set their distances to \(\infty\) except for \(((015, 015)\) = 0. We also push 015 into the queue \(PQ_2\), similarly to the basic search algorithm. When the traversal meets pivot 017, we take the following steps. We load the shortest path tree \(T(017)\) and update distance \(d(016, v)\) for each vertex \(v\) in \(RS_2\).

6.2. Pivot selection

In this section, we discuss how to select pivots to speed up the query. Obviously, more pivots can reduce the search space more in the pivot-based search algorithm. On the other hand, more pivots lead to more space cost. Theorem 6.2 tells us that it is NP-hard to select the number of pivots that maximizes the cover ratio (Definition 6.2) under a limited amount of storage cost.

Definition 6.2. Given a shortest path tree \(T(v)\) rooted at \(v\) and a set of pivots \(PV\), the covered ratio is

\[
cr(T(v)) = \frac{|\{v' | v' is covered by p in T and p \in PV\}|}{|V(G)|}
\]

Theorem 6.2. Given a constant \(M\), finding a pivot set \(PV\) to maximize \(\sum_{v \in V(G)} cr(T(v))\) is an NP-hard problem, where \(|PV| = M\) and \(T(v)\) denotes the shortest path tree rooted at \(v\).

Proof. First, we define a universal set \(U = V(G) \times V(G)\). Second, for each vertex \(v\), we define \(CS(v)\) as follows: 1) \(CS(v) \subseteq U\); 2) \((v', v') \in CS(v)\) if and only if \(v'\) is covered by \(v\) in the shortest path tree of \(v'\). Finding the optimal pivot set is then equivalent to selecting \(M\)
vertices \( \{v_1, v_2, \ldots, v_M\} \) to maximize \( \sum_{i=1}^{v_M} CS(v_i) \). Obviously, this is equivalent to the set cover problem and is \( \text{NP}-\text{hard}. \)

Hence, we must use some heuristic strategies to select pivots. We study the effect of different vertex measures in pivot selection, such as vertex degrees and betweenness. Our experiments confirm that a high-degree strategy (i.e., selecting \( M \) vertices with the top-\( M \) highest degrees) always leads to fast query performance. Note that \( M \) determines the overall space cost. We assume that parameter \( M \) is given by users according to the available space size.

6.3. Further optimization

As the pivot-based search algorithm runs, more and more exact distances from the source to most other vertices have been computed. As a result, the effect of the pivots’ shortest path tree becomes smaller and smaller. At the last moment of the pivot-based search, only a few vertices’ distances remain uncomputed. Then, when we pop a pivot, it would be better to put its neighbors into the queue rather than load its shortest path tree.

Therefore, we can count the number of update operations, \( \text{Count}_{\text{update}} \), when we utilize the current pivot’s shortest path tree (in Lines 7–8 of Algorithm 3). The cost of using the next pivot’s shortest path tree can then be estimated as follows:

\[
\text{Cost}_{\text{update}} = \text{Count}_{\text{update}} \times \text{Cost}_{\text{CPU}}
\]

where \( \text{Cost}_{\text{CPU}} \) is the average CPU cost of a distance update operation.

In addition, we suppose that \( \text{Cost}_{\text{IO}} \) is the average I/O cost to load and scan a pivot’s shortest path tree. We can then use the following two conditions to check whether to continue to load and scan a pivot’s shortest path tree: 1) if \( \text{Cost}_{\text{update}} \leq \text{Cost}_{\text{IO}} \), we continue to load and scan the next pivot’s shortest path tree to update the tentative distances, and 2) if \( \text{Cost}_{\text{update}} > \text{Cost}_{\text{IO}} \), we end loading and scanning the next pivot’s shortest path tree to update the tentative distances.

Note that both \( \text{Cost}_{\text{CPU}} \) and \( \text{Cost}_{\text{IO}} \) are constant if the machines and RDF datasets are given. Therefore, we can set these two values offline and only need to count the number of update operations online.

7. Experiments

In this section, we evaluate our approaches using three large real RDF graphs, DBLP, Yago, and DBPedia.

For the effectiveness study, we compare our methods with the classical keyword search algorithm BANKS [5] on both Yago and DBPedia. Furthermore, because each resource in DBPedia is annotated by Wikipedia documents, we design a stronger baseline called Annotated SPARQL for DBPedia. Annotated SPARQL is similar to the approach discussed in [16]. It first determines all the matches of the SPARQL query, then ranks these matches by how closely the corresponding Wikipedia documents match the keywords. Note that, except for DBPedia, most current RDF datasets do not provide such documents for annotating the resources. Hence, we perform annotated SPARQL experiments on DBPedia. For other RDF datasets, although we can crawl some pages to annotate their entities, that is beyond the scope of this paper.

For the efficiency study, because there is no existing method for SK queries, we evaluate our approaches with a baseline method, i.e., exhaustive computing, which was introduced in Section 1. We call our basic search algorithm (shown in Algorithm 1) Basic Search, while we call our pivot-based search algorithm (shown in Algorithm 3) Pivot-Based Search.

7.1. Datasets and setup

We use three real-world RDF datasets, DBLP, Yago and DBPedia in our experiments. The details about the two datasets are as follows.

**DBLP.** DBLP contains bibliographic information for computer science publications [17]. It contains 8,381,852 RDF triples. We define five sample SK queries for DBLP and show two of them in Table 3 as a case study.

**Yago.** Yago extracts facts from Wikipedia and integrates them with the WordNet thesaurus [11]. It has 18,343,368 triples. We define eight sample SK queries for Yago and show two of them as a case study in Table 4.

**DBPedia & QALD.** DBPedia is an RDF dataset extracted from Wikipedia [18]. It contains 54,440,096 triples. QALD2 is an evaluation campaign on question answering over linked data. It is co-located with the ESWC 2012. In this campaign, the committee provides some questions and each question is annotated with some recommended keywords and the answers that these queries retrieve. Note that, some questions in QALD are so simple that they can map to a SPARQL query with only one edge. It is unnecessary to split these simple questions into a SPARQL query and some keywords. Thus, we only select 10 non-aggregation complex queries from QALD for evaluation. Two of them are as shown in Table 5 as a case study.

Our experiments were conducted on a machine with a 2.4 Ghz Core 2 Duo processor and 80G RAM memory. All experiments were implemented in Java. We used Berkeley DB3 to store the indices. The code and data sets are available on GitHub.4

7.2. Effectiveness study

In this section, we compare our methods with the classical keyword search algorithm BANKS [5] on DBLP and Yago to show the effectiveness of our methods. Furthermore, because each resource in DBPedia is annotated by Wikipedia documents, we designed a stronger baseline method called “Annotated SPARQL” for DBPedia. Note that because the distance-based optimization only improves the efficiency, the effectiveness of the basic search and the pivot-based search is the same.

\[ \text{http://greenentacle.technak.uni-bielefeld.de/~cunger/qald/index.php?x=challenges&q=2}. \]

\[ \text{http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html}. \]

\[ \text{https://github.com/bnu05pp/SKQueryProcessing}. \]
Table 4  
Sample Yago queries for the case study.

<table>
<thead>
<tr>
<th>Query Semantic</th>
<th>SK Query</th>
<th>SPARQL</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1: Which actors/actresses played in Philadelphia are mostly related to Academy Award and Golden Globe Award?</td>
<td>Select ?p where { ?p type actor; ?p actedIn ?f; ?f label &quot;Philadelphia&quot;;}</td>
<td>Academy Award, Golden Globe Award</td>
<td></td>
</tr>
<tr>
<td>Q2: Which Turing Award winners in the field of database are mostly related to Toronto?</td>
<td>Select ?p where { ?p hasWonPrize ?a; ?a label &quot;Turing Award&quot;;}</td>
<td>Toronto, database</td>
<td></td>
</tr>
</tbody>
</table>

Table 5  
Sample QALD queries on DBPedia for the case study.

<table>
<thead>
<tr>
<th>Query Semantic</th>
<th>SK Query</th>
<th>SPARQL</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1: Which states of Germany are governed by the Social Democratic Party?</td>
<td>Select ?s where { ?s country ?g; ?g name &quot;Germany&quot;;}</td>
<td>Social Democratic Party</td>
<td></td>
</tr>
<tr>
<td>Q2: Which monarchs of the United Kingdom were married to a German?</td>
<td>Select ?u where { ?u spouse ?s; ?s birthPlace ?c; ?c name &quot;Germany&quot;;}</td>
<td>United Kingdom, monarch</td>
<td></td>
</tr>
</tbody>
</table>

Table 6  
Effectiveness results for sample DBLP queries.

<table>
<thead>
<tr>
<th>Top-3 Results of SK Query</th>
<th>Top-3 Results of BANKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>David J DeWitt</td>
<td>Katsumi Tanaka</td>
</tr>
<tr>
<td>Jeffrey Xu Yu</td>
<td>Mong-Li Lee</td>
</tr>
<tr>
<td>Tok Wang Ling</td>
<td>Reda Alhaj</td>
</tr>
</tbody>
</table>

7.2.1. Case study  
We show six sample queries in Tables 3–5 for the case study.  
DBLP. Let us consider the two sample queries in DBLP. The top-3 results answered by the SK query and BANKS on DBLP are shown in Table 6.

Q1: Which researchers on keyword search published papers in VLDB 2004 and DEXA 2005?  
The three results returned in SK query are three researchers named “David J DeWitt”, “Jeffrey Xu Yu” and “Tok Wang Ling”. All of them published papers in both VLDB 2004 and DEXA 2005. Furthermore, they did research about keyword search. However, the first three results returned by the traditional keyword search are “Katsumi Tanaka”, “Mong-Li Lee” and “Reda Alhaj”. Although all of them are interested in keyword search, none of them had published a paper in VLDB 2004.  
Q2: Which papers in KDD 2005 about concept-drifting are written by Jiawei Han?  
The first result returned by the SK query is a paper titled “Mining concept-drifting data streams using ensemble classifiers”. This paper was written by Jiawei Han and published in KDD 2005. This is a paper closely related to concept-drifting and is the best answer to query Q2. The other two results of SK queries are two more papers written by Jiawei Han and published in KDD 2005. In contrast, the first two results returned by the traditional keyword search are two papers about concept-drifting, but neither of them was published in KDD 2005 nor written by Jiawei Han. The third result of the traditional keyword search is a researcher who is even more unrelated to the query.

Yago. Let us consider the two sample queries in Yago. The top-3 results answered by the SK query and BANKS over Yago are shown in Table 7.

Q1: Which actors/actresses played in Philadelphia are mostly related to Academy Award and Golden Globe Award?  
We had analyzed query Q3 in Section 1. For comparison, we use the keywords {actors, actresses, Philadelphia, Academy Award, Golden Globe Award} for the keyword search. Generally, an SK query returns more reasonable answers than the traditional keyword search. In contrast, the first two results returned by the traditional keyword search are “Grace Kelly” and “George Cukor”. Grace Kelly lived in Philadelphia, and George Cukor is also an actor who directed the film, The Philadelphia Story, in 1940.

Q2: Which Turing Award winners in the field of database are mostly related to Toronto?  
The first result returned in SK query is “Stephen Cook”. As we know, Stephen Cook is a professor at the University of Toronto. He won the Turing award for his contributions to complexity theory. This is the best answer to query Q2. The second answer is “William Kahan”. Prof. William Kahan was born in Toronto and won the Turing award for his contributions to the numerical analysis algorithm. The third one is “Kenneth E. Iverson”. Prof. Kenneth E. Iverson also received the Turing Award. He died in Toronto.

In contrast, the first two results returned by traditional keyword search are “English Language” and “Princeton University”. Obviously, they are non-informative results. Here, the keywords used for the keyword search are [Turing Award, winners, Toronto].

DBPedia & QALD. Let us consider the two sample QALD queries over DBPedia. The top-3 results answered by BANKS, the SK query and Annotated SPARQL over DBPedia are shown in Table 8.

Q3: Which states of Germany are governed by the Social Democratic Party?  
The first three results returned in SK query are three places in Germany and governed by the Social Democratic Party. How-
ever, the first three results returned by the annotated SPARQL are three members of the Social Democratic Party in Germany. The first three results returned by the traditional keyword search are three places far from Germany. Hence, the results of the SK queries are more informative than the other two methods. Here, the keywords used for the keyword search are {state, Germany, govern, SocialDemocraticParty}, which are given in QALD.

$Q_6$: Which monarchs of the United Kingdom were married to a German?

The first result of both the SK query and annotated SPARQL are William IV of the United Kingdom, which is the best answer. The other two results of SK query are still two royals in Europe. However, the third result of annotated SPARQL is an European nation. In addition, the first three results returned by traditional keyword search are non-informative results. Here, the keywords for the keyword search are {United Kingdom, monarch, married, German}, which are also given in QALD.

### 7.2.2. NDCG@$k$ over Yago and DBLP

To quantify the effectiveness of the SK query, we evaluated the normalized discounted cumulative gain (NDCG) [19] of both SK query and the keyword search. Because there are no golden standards, we invited 10 volunteers to judge the result quality. Specifically, we asked each volunteer to rate the goodness of the results returned by the SK query and keyword search method. The scores ranged between 1 and 5. Higher scores indicate better results.

Table 9 reports the NDCG@$k$ values when $k$ was varied from 3 to 10 in both Yago and DBLP. The SK query outperforms the traditional keyword search by 20%–50%. Furthermore, we find that the gap in Yago is larger than that in DBLP. The reason is that Yago has a more complex schema than DBLP. Thus, keywords may result in more ambiguity in Yago than in DBLP. This indicates that the superiority of SK query is more pronounced in semantic-rich data.

### 7.2.3. MAP over DBPedia

Because QALD provides the standard answers of each query, we used the mean average precision (MAP) [20] to compare the SK query with BANKS and Annotated SPARQL.

Table 10 reports the MAP values of the ten QALD queries. Both the SK query and annotated SPARQL outperform the traditional keyword search by an order of magnitude. The MAP value of the annotated SPARQL is smaller than that of the SK query. This is because that the annotated SPARQL can do well when the documents associated with the matches contains the keywords. In other words, it works only when the relation between the matches and the keywords is explicit. However, in practice, the relation between the matches and the keywords is often implicit. In this case, the SK query performs better.

### 7.3. Efficiency study

In this section, we evaluate the efficiency of the SK query on large real graphs. Here, the default number of returned results was set to 10.

#### 7.3.1. Pruning effect of the structural index

Based on the indices introduced in Section 4, we can avoid calls Algorithm 2 for graph matching by pruning many unsatisfied vertices. Moreover, the vertices that are too far away to be in a final answer can be safely pruned. For this experiment, we report the pruning efficiency of our structural index. We compare the number of graph matching operations that our search algorithms accessed with those that the exhaustive computing approach accessed. Note that the pivot-based algorithm only speeds up the traversal, as discussed in Section 6, so the number of graph matching operations in the basic and pivot-based searches is the same.

Tables 11–13 show the number of graph matching operations on DBLP, Yago and DBPedia, respectively. The number of graph matching operations in advanced backward search is not less than it is for the basic backward search. In most cases, we avoid a large number of graph matching operations.

#### 7.3.2. Evaluation of pivot selection methods

In this experiment, we used DBLP and Yago to test the effect of different pivot selection methods (see Section 6.2). The methods consist of random selection (denoted as Random), highest degree-based selection (finding the M highest degree vertices as pivots, denoted as HD), and largest betweenness-based selection (finding M largest degree of betweenness as pivots, denoted as HB). Here, we set M to be a default value of 500. The effect of M is evaluated shortly. Fig. 11 shows that both the degree-based selection method and the betweenness-based selection method lead to much faster query performance than the random selection method. The gap between the degree-based method and the betweenness-based method is not large. Because it is much cheaper to find the
Table 11  
Number of graph matching operations on DBLP.

<table>
<thead>
<tr>
<th></th>
<th>$Q_1$</th>
<th>$Q_2$</th>
<th>$Q_3$</th>
<th>$Q_4$</th>
<th>$Q_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive Computing</td>
<td>292268</td>
<td>21885</td>
<td>254674</td>
<td>872426</td>
<td>2747</td>
</tr>
<tr>
<td>Basic Search/Pivot-based Search</td>
<td>354</td>
<td>5</td>
<td>1684</td>
<td>669</td>
<td>1548</td>
</tr>
</tbody>
</table>

Table 12  
Number of graph matching operations on Yago.

<table>
<thead>
<tr>
<th></th>
<th>$Q_1$</th>
<th>$Q_2$</th>
<th>$Q_3$</th>
<th>$Q_4$</th>
<th>$Q_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive Computing</td>
<td>600283</td>
<td>563736</td>
<td>301</td>
<td>167958</td>
<td>231210</td>
</tr>
<tr>
<td>Basic Search/Pivot-based Search</td>
<td>6</td>
<td>55</td>
<td>209</td>
<td>5414</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 13  
Number of graph matching operations on DBPedia.

<table>
<thead>
<tr>
<th></th>
<th>$Q_1$</th>
<th>$Q_2$</th>
<th>$Q_3$</th>
<th>$Q_4$</th>
<th>$Q_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive Computing</td>
<td>31083</td>
<td>136777</td>
<td>19904</td>
<td>847</td>
<td>19454</td>
</tr>
<tr>
<td>Basic Search/Pivot-based Search</td>
<td>13824</td>
<td>3941</td>
<td>4769</td>
<td>847</td>
<td>40</td>
</tr>
<tr>
<td>Exhaustive Computing</td>
<td>40302</td>
<td>5076</td>
<td>23422</td>
<td>16079</td>
<td>2786</td>
</tr>
<tr>
<td>Basic Search/Pivot-based Search</td>
<td>89</td>
<td>18</td>
<td>23422</td>
<td>16079</td>
<td>1</td>
</tr>
</tbody>
</table>

![Graph matching performance](a_results_on_dblp.png)  
![Graph matching performance](b_results_on_yago.png)  

**Fig. 12.** Search performance with respect to number of pivots.

<table>
<thead>
<tr>
<th></th>
<th>Structural Index</th>
<th>Distance Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>Index Construction Time(s)</td>
<td>143.4</td>
</tr>
<tr>
<td></td>
<td>Index Size(GB)</td>
<td>0.6</td>
</tr>
<tr>
<td>Yago</td>
<td>Index Construction Time(s)</td>
<td>248.3</td>
</tr>
<tr>
<td></td>
<td>Index Size(GB)</td>
<td>1.0</td>
</tr>
<tr>
<td>DBPedia</td>
<td>Index Construction Time(s)</td>
<td>2345.9</td>
</tr>
<tr>
<td></td>
<td>Index Size(GB)</td>
<td>4.5</td>
</tr>
</tbody>
</table>

M vertices with the highest degrees than those with the largest betweenness, we use degree-based selection.

7.3.3. Evaluation of pivot numbers

In this section, we use DBLP and Yago to test how many pivots we need to select to benefit the online query. The default pivot selection method is the highest-degree selection. Fig. 12 shows that query response times decrease for all methods when the pivot number was varied from 100 to 500. Obviously, the more pivots we choose, the smaller the query response time is. However, more pivots lead to larger space cost. In this paper, we set the number of pivots to 500.

7.3.4. Offline performance

We report the index size and index construction time in Table 14. Because our structural index is based on efficient sequential pattern mining, we can finish the structural index construction in several minutes. Similarly, for our pivots-based index, we only need to compute some shortest path trees offline, which can also be finished within acceptable response times.

7.3.5. Online performance

In this section, we evaluate the efficiency of our methods. Fig. 13 shows the query times of the three methods.

As shown in Fig. 13, our method outperforms the baseline method by two or more times in most cases. Especially for $Q_3$ on Yago and $Q_2$, $Q_5$ on DBLP, our method only takes a fifth of the time as the exhaustive computing approach. This is because the matches of these SPARQLs are close to the vertices containing the keywords. Thus, the query processing can terminate quickly.

Note that because our inverted index for the keywords are stored on disk, keyword mapping takes a long time and takes up a large part of the total time. Hence, it is difficult for our method to improve efficiency too much.

8. Related work

There have been many studies to investigate the processing of SPARQL queries, such as [13,21–26]. Some [22,25] store the RDF triples into an RDBMS and answer the SPARQL via join operations. RDF-3X [13,23] and Hexastore [21] create indexes for each permutation of subject, predicate and object. Because an RDF dataset can also be modeled as a graph, gStore [24] and AMbER [26] answer SPARQL in an RDF dataset by finding the subgraph matches over an RDF graph. AMbER and gStore design a subgraph match algorithm similar to VF2 [27] to answer a SPARQL query. VF2 [27] is an early effort at subgraph isomorphism checking. It starts with a vertex and explores vertices connected to the already matched query vertices one by one.

For keyword search, existing keyword search techniques over RDF graphs can be classified into the following two categories. The first kind of methods [28–31] interpret keywords as SPARQL queries and then retrieve results by invoking existing SPARQL query engines. The other kind of methods aim to find the small-size substructures (in RDF graphs) that contain all keywords. The top-k substructures, like trees [5–9,32,32,33], cliques [10] or other
patterns defined over the RDF graphs [34], are computed on the basis of a scoring function and are returned to users.

There are also many approaches mining some frequent patterns to build indices in graph database [15,35,36]. Among these works, gIndex [15] and gSpan [35] can be applied to small graphs in a database of multiple graphs, but do not support mining patterns in a single graph. GADDI [36] tries to find all the matches of a query graph in a large graph, but it can only support a graph with thousands of vertices, while recent RDF data graphs may have hundreds of thousands of entities.

To the best of our knowledge, although there exist a few previous works [16,37] on hybrid queries combining SPARQL and keywords, there is no existing work on the SK query defined as above. Elbassuoni et al. [37] assume that each RDF triple may have associated text passages. Then, they extend the triple patterns in SPARQL with keyword conditions. Moreover, CE [16] assumes that each resource is associated with a document. It then extends the variables in SPARQL using keyword conditions. Nonetheless, most current RDF datasets do not provide either text passages to annotate triples or documents to annotate resources. In summary, neither of these methods can handle our example queries. In addition, the SK query that we define can apply to most existing RDF datasets.

In addition, in [38], the authors define a new query language that blends keyword search with structured query processing [39]. utilizes some given kinds of SPARQL to improve the result of object retrieval. Moreover, Bhagdev et al. [40] and Bikakis et al. [41] try to extend keyword search with semantics. Zou et al. [42,43] translate natural language questions into SPARQL queries.

9. Conclusions

In this paper, we proposed a new kind of query (the SK query) that integrates SPARQL and keywords. To handle this kind of query, we first introduced a basic method based on backward search. However, this basic solution faces several performance issues. Hence, we built a structural index and a distance-based index. Our structural index is based on frequent star patterns in the RDF data, and our distance-based index is based on the shortest path trees of selected pivots in the RDF graph. Using the indices, we propose an advanced strategy to deal with SK queries. Finally, using three real RDF datasets, we demonstrated that our method can outperform the baseline both with respect to effectiveness and efficiency.

Acknowledgments

Peng Peng’s work is supported by the Fundamental Research Funds for the Central Universities. Lei Zou is supported by the National Key Research and Development Program of China under grant 2016YFB1000603, NSFC under grant 6162201, 61532010 and 61370055. This work is also partially supported by the National Science Foundation of China under Grant No.61472131, 61272546; Science and Technology Key Projects of Hunan Province (2015 TP1004, 2016JC2012).

Appendix A. Queries in experiments

Tables A.15 and A.16 show all of our sample queries over Yago and DBLP. Here, since our institution is in China, most volunteers that we invite are Chinese. Hence, some sample queries are about China.

For more reasonable experiments, so we also sample 10 non-aggregation QALD queries over DBPedia to evaluate our method. All QALD queries over DBPedia are shown in Table A.17.
Table A.15
Sample queries over DBLP.

<table>
<thead>
<tr>
<th>Query Semantic</th>
<th>SK Query</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2 Which papers in KDD 2005 about concept-drifting are written by Jiawei Han?</td>
<td>Select ?paper where { ?paper year 2003 ?paper booktitle KDD ?person creator ?person ?person name Jiawei Han} concept-drifting</td>
<td></td>
</tr>
<tr>
<td>Q5 Which two researchers did research about skyline and coauthored a paper in VLDB 2005?</td>
<td>Select ?person1, ?person2 where { ?paper year &quot;2005&quot; ; ?paper booktitle &quot;VLDB&quot; ; ?paper creator ?person1 ; ?paper creator ?person2 } Skyline</td>
<td></td>
</tr>
</tbody>
</table>

Table A.16
Sample queries over Yago.

<table>
<thead>
<tr>
<th>Query Semantic</th>
<th>SK Query</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 Which actresses played in Philadelphia are mostly related to Academy Award and Golden Globe Award?</td>
<td>Select ?person where { ?person year 2017 ; ?person label &quot;Philadephia&quot; ; ?person Academy Award , ?person Golden Globe Award} Academy Award, Golden Globe Award</td>
<td></td>
</tr>
<tr>
<td>Q2 Which Turing Award winners in the field of database are mostly related to Toronto?</td>
<td>Select ?person where { ?person type scientific; ?person hasWonPrize 7a; ?person capital &quot;Turing Award&quot;} Toronto, database</td>
<td></td>
</tr>
<tr>
<td>Q3 Which Microsoft's products are about SDK?</td>
<td>Select ?category where { ?category type company; ?category label &quot;Microsoft&quot;; ?category created 7c} SDK</td>
<td></td>
</tr>
<tr>
<td>Q5 Which top members of Communist Party of China are related Kissinger?</td>
<td>Select ?person where { ?person isAffiliatedTo ?organization; ?person label &quot;Communist Party of China&quot;; ?person type Politician;} Kissinger</td>
<td></td>
</tr>
<tr>
<td>Q6 Whose father was United States Army generals and took part in Normandy Invasion?</td>
<td>Select ?person where ?parent ?person ?person2 ; ?person2 ?person UnitedStatesArmyGenerals ; ?person type UnitedStatesArmyGenerals ;</td>
<td>Normandy Invasion</td>
</tr>
<tr>
<td>Q7 Which state generated a Los Angeles Lakers player that relate to Eagle, Colorado?</td>
<td>Select ?person where { ?person bornIn ?place; ?person locationIn ?state; ?state type Los Angeles Lakers Players ;} Eagle Colorado</td>
<td></td>
</tr>
<tr>
<td>Q8 Which participants of People's National Congress did graduate from universities in Beijing?</td>
<td>Select ?person where { ?person graduatedFrom ?university; ?university type UniversitiesInBeijing;} People's National Congress</td>
<td></td>
</tr>
</tbody>
</table>

Table A.17
Sample QALD queries over DBPedia.

<table>
<thead>
<tr>
<th>Query Semantic</th>
<th>SK Query</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 Which states of Germany are governed by the Social Democratic Party?</td>
<td>Select ?state where { ?state country ?region; ?region name &quot;Germany&quot;;} Social Democratic Party</td>
<td></td>
</tr>
<tr>
<td>Q2 Which monarchs of the United Kingdom were married to a German?</td>
<td>Select ?person where { ?person spouse ?person2; ?person2 birthPlace ?place; ?place name &quot;Germany&quot;;} United Kingdom, monarch</td>
<td></td>
</tr>
<tr>
<td>Q3 Which capitals in Europe were host cities of the summer olympic games?</td>
<td>Select ?capital where { ?capital type Country; ?capital capital ?city} Olympic games, Europe</td>
<td></td>
</tr>
<tr>
<td>Q6 List all episodes of the first season of the HBO television series The Sopranos!</td>
<td>Select ?episode where { ?episode series &quot;The Sopranos&quot;; ?episode season 1} HBO, first</td>
<td>HBO, first</td>
</tr>
<tr>
<td>Q7 In which films directed by Garry Marshall was Julia Roberts starring?</td>
<td>Select ?film where { ?film type Film; ?film director ?director; ?director name &quot;Garry Marshall&quot;;} Julia Roberts</td>
<td></td>
</tr>
<tr>
<td>Q8 Which software has been developed by organizations founded in California?</td>
<td>Select ?software where { ?software type Organisation; ?software developer ?person; ?person locationIn California} California</td>
<td>gold, mineral</td>
</tr>
<tr>
<td>Q9 Which U.S. states possess gold minerals?</td>
<td>Select ?state where { ?state type Place; ?state country ?country; ?country name &quot;the United States&quot;;}</td>
<td></td>
</tr>
</tbody>
</table>

References


[14] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, M. Hsu, Prefix-


[17] M. Ley, P. Reuther, Maintaining an online bibliographical database: the prob-


[22] D.J. Abadi, A. Marcus, S. Madden, K. Hollenbach, SW-Store: a vertically par-

[23] T. Neumann, G. Weikum, X-RDF-JX: fast querying, high update rates, and con-
sistency for RDF databases, PVLDB 3 (1) (2010) 256–263.


[27] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, A (sub)graph isomorphism algo-


[29] T. Tran, H. Wang, S. Rudolph, P. Cimiano, Top-k exploration of query candi-


[36] S. Zhang, S. Li, J. Yang, Gaddi: distance index based subgraph matching in bio-


[38] J. Pound, I.F. Ilyas, G.E. Weddell, Expressive and flexible access to web-ex-

[39] A. Tonon, G. Demartini, P. Cudré-Mauroux, Combining inverted indices and


[42] L. Zou, R. Huang, H. Wang, J.X. Yu, W. He, D. Zhao, Natural language ques-