Answering Top-K Query Combined Keywords and Structural Queries on RDF Graphs

Peng Peng[1], Lei Zou[2], Zheng Qin[1]

[1]*Hunan University, Changsha, China*

[2]*Peking University, 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 *S*PARQL-*K*eyword (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.

Keywords: SPARQL, Keyword Search, RDF Graph

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 ⟨subject, predicate, object⟩. An RDF repository can be represented as a graph, where subjects and objects are

Email address: [1]{hnu16pp,zqin}@hnu.edu.cn,[2]zoulei@pku.edu.cn (Peng Peng[1], Lei Zou[2], Zheng Qin[1])

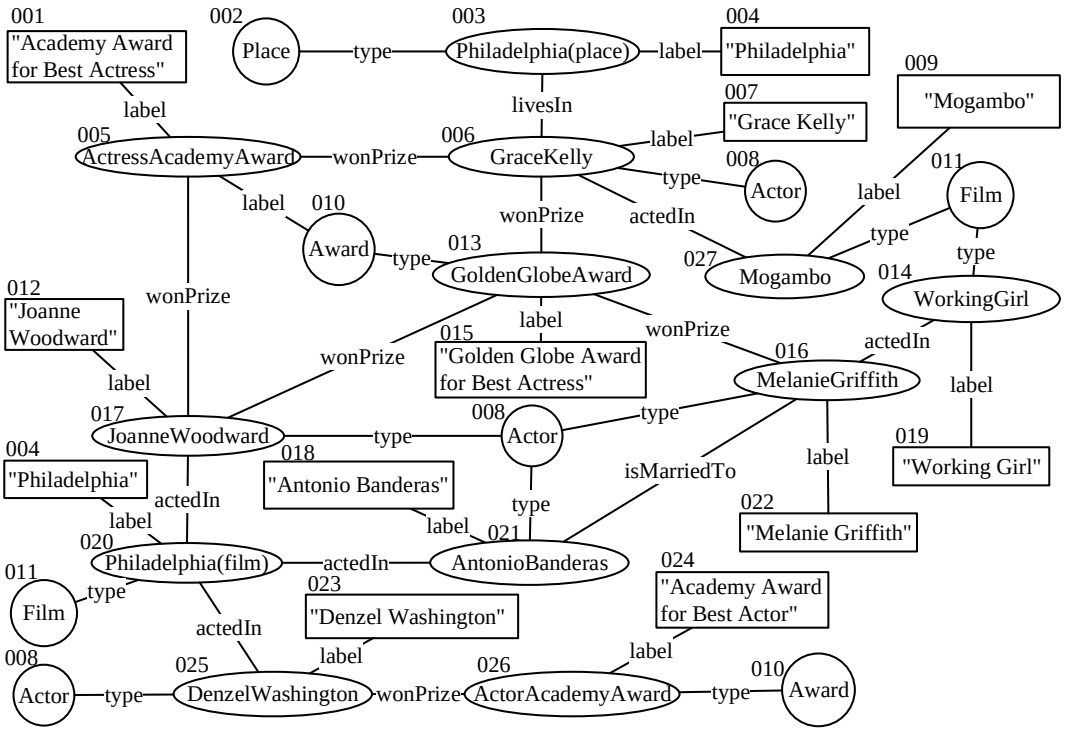


Figure 1: Example RDF Graph

vertices connected by labeled edges (i.e., predicates). Figure 1 shows an example RDF dataset and the corresponding RDF graph, which is a part of the well-known knowledge base Yago [29]. 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.

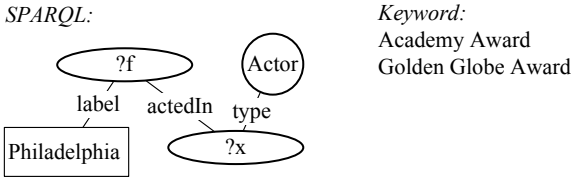


Figure 2: Example SPARQL-Keyword Query

As we know, the SPARQL query language is a standard way to access RDF data and is based on the subgraph (homomorphism) match semantic [27]. Figure 3(a) shows an example of a SPARQL query, and its corresponding query graph is

shown in Figure 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".*

SPARQL:
Select ?a where{
?a type Actor.
?a actedIn Philadelphia(film).
Philadelphia(film) type Film}

(a) Q_1

Select ?a where{
?a ?p ?str.
?a type Actor.
?a actedIn Philadelphia(film).
Philadelphia(film) type Film.
FILTER regex(?str, "(Academy Award)|
(Golden Globe Award)")}

(b) Q_2

Figure 3: Example SPARQL Queries

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 Figure 3. More typographic or linguistic distances, such as string edit distance [10] and Google similarity distance [7], 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. Figure 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". 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 [2, 19, 16, 11, 21, 20] 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.

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 *S*PARQL-*K*eyword (SK) query, shorted as 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 $\langle Q, q \rangle$. The SPARQL query graph Q is given in Figure 2, where the keywords are q = {Academy Award, Golden Globe Award}. Figure 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 strength than the two-hop path through 008. Therefore, following the intuition of TF-IDF for measuring the word importance in a corpus, we propose *predicate salience* (see Section 2) to evaluate relation strengths.

Another challenge of this problem is the search efficiency. A naÃŕve exhaustive-computing strategy works as follows: we first find all subgraph matches of Q (in

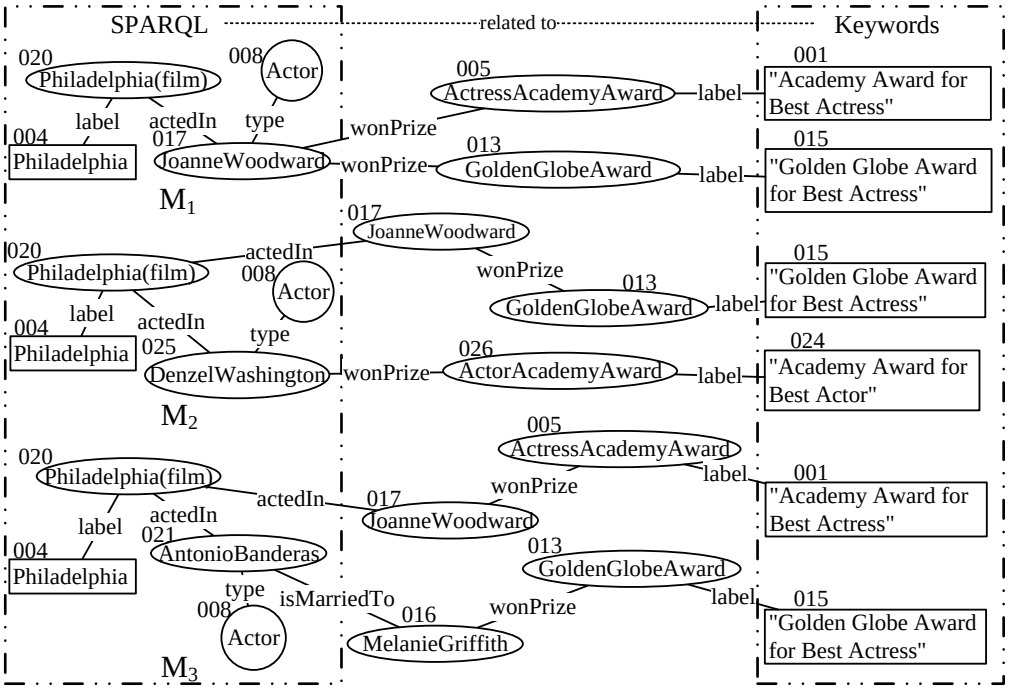


Figure 4: SK Query Results

RDF graph G) using existing techniques. Then, we compute the shortest path distances between these subgraph matches and the vertices containing keywords on the fly. Finally, the matches with the shortest distances to the keywords are returned as answers. Obviously, this is an inefficient solution. 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 exhaustive computing, we only find matches for SPARQL query Q progressively 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 keywords, 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 utilize 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-based 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 effectiveness (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 presented 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 corresponds 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.

Definition 2.1. *An RDF data graph G is denoted as $\langle V(G), E(G), L\rangle$, where (1) $V(G) = V_L \cup V_E \cup V_C$ is the set of vertices in RDF graph G(V_L, V_E and V_C 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 (*S*PARQL & *K*eyword) *query is a pair $\langle Q, q\rangle$, where Q is a SPARQL query graph, and q is a set of keywords $\{w_1, w_2, ..., w_n\}$.*

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

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

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

$$Cost(r) = Cost_{content}(r) + Cost_{structure}(r),$$

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

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

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

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 [33] or Google similarity distance [7] 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 $\langle M, \{v_1, v_2, ..., v_n\}\rangle$ for an SK query $\langle Q, q\rangle$, the distance between match M and vertex v_i ($i = 1, ..., n$) is defined as follows.*

$$d(M, v_i) = MIN_{v \in M}\{d(v, v_i)\}$$

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

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

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

(**Problem Definition**) *Given an SK query $\langle Q, q\rangle$ 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 Figure 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 Figure 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.

Predicate	Predicate Salience
actedIn	0.296
isMarriedTo	0.074
label	0.852
livesIn	0.074
type	0.593
wonPrize	0.259

Table 1: Weights of Predicates in the Example RDF Graph

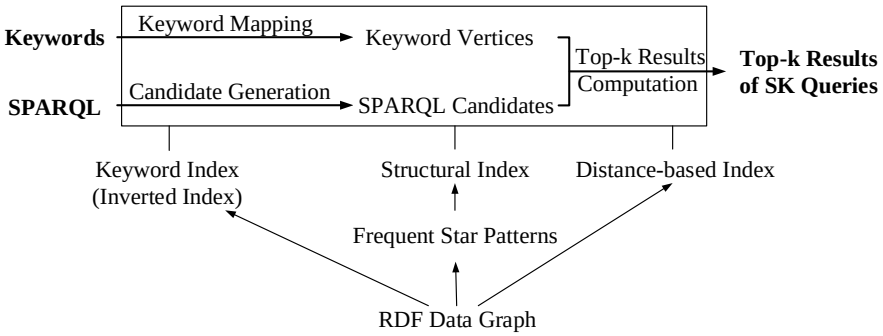


Figure 5: Overview of Our Approach

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 ho-

momorphism is not efficient because of its high complexity [15]. 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 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 [24].

We propose a sequential pattern mining-based method to find frequent star 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 $\langle actedIn, actedIn, actedIn, name, type\rangle$. 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 [26] 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 $\langle actedIn, type\rangle$ and $\langle actedIn, type, wonPrize\rangle$ are two frequent sequential patterns. It is easy to see that a sequential pattern always corresponds to one star pattern, as shown in Figure 6. For ease of presentation, we use the terms "sequential patterns" and "star patterns" interchangeably in the following discussion.

Vertex	Predicate Sequence
Philadelphia(film)	$\langle actedIn, actedIn, actedIn, label, type\rangle$
JoanneWoodward	$\langle actedIn, actedIn, label, type, wonPrize\rangle$
AntonioBanderas	$\langle actedIn, label, type, wonPrize, wonPrize\rangle$
...	...

Table 2: Examples of Predicate Sequences

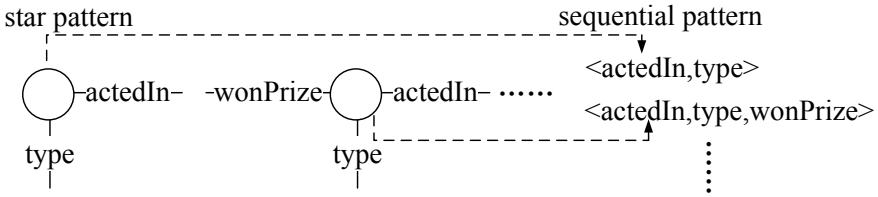


Figure 6: Example Star Pattern

For each frequent star pattern S, we maintain an inverted list $L(S) = \{v|S$ occurs in v's adjacent edge sequence$\}$. Obviously, if we use all frequent stars as the index elements, the space cost is very large. Thus, inspired by gIndex [39], 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:*

$$\gamma(S) = \frac{|L(S)|}{|\cap_{S' \subset S} L(S')|}$$

where $S' \subset S$ 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 $S' \subset S$. 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 γ_{max}, and we only choose stars S such that $\gamma(S) \leq \gamma_{max}$. 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-negatives, 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.*

$$v \notin \bigcup\nolimits_{S \in F \wedge S \in Q} L(S),$$

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 $v \notin \bigcup_{S \in F \wedge S \in Q} L(S)$, 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. □

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 "?a" of the SPARQL query in Figure 2 has the predicate sequence $\langle actedIn, type \rangle$. 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.*

$$v \notin \bigcup\nolimits_{S \in F \wedge S \in Q} L(S),$$

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 [2]. 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_i denote all literal vertices (in RDF graph G) containing keyword w_i.

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

For graph exploration, we maintain a priority queue PQ_i for each keyword w_i. Each element in PQ_i is represented as $(v, p, |p|)$, where v is a vertex ID, p is a path between v and a vertex in V_i and $|p|$ denotes the path distance. All elements in PQ_i are sorted in non-descending order of $|p|$. Each keyword w_i is also associated with a result set RS_i. To keep track of information related to each vertex v, we associate v with a vector $d[v]$. If a vertex v is in RS_i ($i = 1, ..., n$), the shortest

Algorithm 1: Search for Top-k Results of SK Queries

Input: RDF data graph G, SK query $\langle Q, q\rangle$, $\{V_1, ..., V_n\}$ where V_i is the set of vertices containing keyword w_i, priority queues $\{PQ_1, ..., PQ_n\}$.

Output: Top-k results R of $\langle Q, q\rangle$.

```
1  for each vertices set V_i do
2      for each vertex v in V_i do
3          Insert (v, ∅, 0) into PQ_i;
4  while not all queues are empty and θ ≥ δ do
5      for i = 1, ..., n do
6          Pop the head of PQ_i (v, p, |p_i|), set d[v][i] = |p_i| and insert it into RS_i;
7          for each adjacent edge vv' to v do
8              if p ∪ vv' 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(vv')) then
12                     Replace (v', p', |p'|) with (v', p ∪ vv', |p_i| + ps(vv')) in PQ_i;
13             else
14                 Insert (v', p ∪ vv', |p| + ps(vv')) 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 δ of top-k results
20         Update the cost of all partially seen matches and δ;
21         Update the lower bound cost θ of all remaining unseen vertices;
22 Return R.
```

path distance is known. In this case, we set $d[v][i] = d(v, w_i)$; otherwise, we set $d[v][i] = null$.

Initially, the exploration starts with a set of vertices containing keywords. For each vertex v containing keyword w_i, an element $(v, \emptyset, 0)$ is created and placed into the queue PQ_i (Line 3 in Algorithm 1). During the search, at each step, we pick a queue PQ_i ($i = 1, ..., n$) 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_i. When a queue head $(v, p, |p|)$ is popped from queue PQ_i, we insert it into result set RS_i and set $d[v][w_i] = |p|$ (Line 6 in Algorithm 1). We prove that the following theorem holds.

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. We 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 $v \notin V_i$ because all vertices in V_i are the first vertices added to set RS_i and $d[v][i] = 0$ at that time.

Because $v \notin V_i$, we also have that $RS_i \neq \emptyset$ 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 RS_i, namely some vertices in V_i, to a vertex in $V(G) - RS_i$, namely v. Let us consider the first vertex v' along p' such that $v' \in V(G) - RS_i$, and let $v'' \in RS_i$ 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 $v'' \in RS_i$. 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' is added to RS_i. Edge $\overline{v'v''}$ 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 $V(G) - RS_i$ 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.□

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 says 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*.*

Assume that $(v, p, |p|)$ is popped from queue PQ_i. For each incident edge $\overline{vv'}$ to v, we obtain a new element $(v', p \cup \overline{vv'}, |p| + ps(\overline{vv'}))$, where $p \cup \overline{vv'}$ denotes appending an edge to p and $ps(\overline{vv'})$ denotes the predicate salience value of the edge label of $\overline{vv'}$. If $p \cup \overline{vv'}$ is not a simple path[1], the element is ignored (Lines 8-9 in Algorithm 1). We then check whether there exists another element $(v', p', |p'|)$ that has an identical vertex v' with the new element $(v', p \cup \overline{vv'}, |p| + ps(\overline{vv'}))$, where $|p'| > |p| + ps(\overline{vv'})$). If so, we delete $(v', p', |p'|)$ from PQ_i and insert $(v', p \cup \overline{vv'}, |p| + ps(\overline{vv'}))$ into PQ_i (Lines 12-13 in Algorithm 1). Otherwise, we ignore the new element (Line 15 in Algorithm 1). If there exists no element $(v, p', |p'|)$, we insert $(v', p \cup \overline{vv'}, |p| + ps(\overline{vv'}))$ into the queue directly (Line 17 in Algorithm 1).

Definition 5.3. ***Fully Seen Vertex, Partially Seen Vertex, and Unseen Vertex***. *Given a vertex* v*, if* v *is seen by all keywords* w_i $(i = 1, ..., n)$*,* v *is called a* fully seen *vertex; if* v *is not a fully seen vertex but has been seen by at least one keyword,* v *is called a* partially seen *vertex; if* v *has not been seen by any keyword,* v *is called an* unseen *vertex.*

At each step, we check whether the vertex just popped from the queue has been seen by all keywords. Specifically, for a popped queue head v, if all dimensions of its vector $d[v]$ are non-null, this indicates that all keywords have seen vertex v, i.e., we know the distance between v and each keyword. In this case, v is a *fully seen* vertex. When we meet a fully seen vertex v, we will employ a subgraph homomorphism algorithm to find matches containing v (Line 18-19 in Algorithm 1). The details are discussed in Section 5.2.

5.2. Generation of SPARQL Matches

When we encounter a fully seen vertex v, this indicates that we know the distance between v and each keyword w_i. The next step is to compute the SPARQL matches containing vertex v, if any. Here, we perform a subgraph homomorphism algorithm to find the subgraph matches (of query Q) containing v.

Generally, we employ a DFS-based *state transformation* algorithm to perform the matching process beginning from a fully seen vertex v (as shown in Algorithm 2). Here, we define the *state* as follows.

[1] A simple path is a path with no repeated vertices.

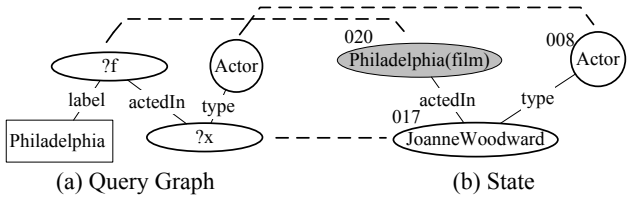


Figure 7: Example State

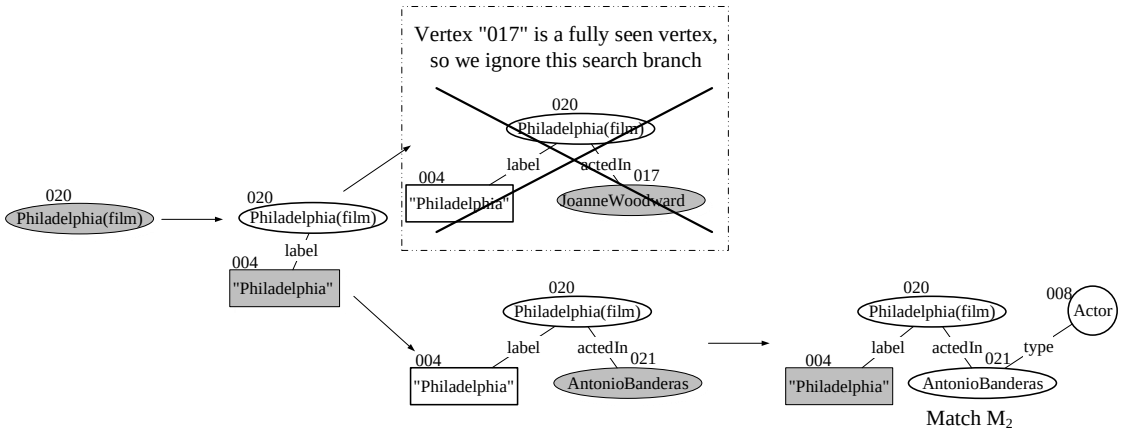


Figure 8: Finding Matches Containing Vertex 020 by Pruning the Search Branch Beginning from 017

Definition 5.4. *Given a SPARQL query graph Q with m vertices $u_1, ..., u_m$, a* state *is a (partial) match of query graph Q.*

For example, Figure 7 shows an example state of the SPARQL query in Figure 2.

In particular, our *state transformation* algorithm is as follows. Assume that v matches vertex u in SPARQL query Q. We first initialize a state with v. We then search the RDF data graph to reach v's neighbor v' corresponding to u' in Q, where u' is one of u's neighbors and edge $\overline{vv'}$ satisfies query edge $\overline{uu'}$. 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

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 ≠ ∅ do
4      Pop the first state s ∈ S;
5      if all edges of Q have been matched in s then
6          Insert s to MS;
7      for each unmatched edge u'u'' that u' has been matched to v' do
8          if u'' has been matched to v'' then
9              if v'v'' ∈ 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 v'v'' can match u'u'' then
16                     Initialize a new state s' and s' = s;
17                     Match u'' with v'';
18                     Push s into S;
19 Return MS.
```

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(film)) and vertex* 017 *(JoanneWoodward) is another a fully seen vertex. As shown in Figure 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 Figure 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.

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*.*

Figure 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.

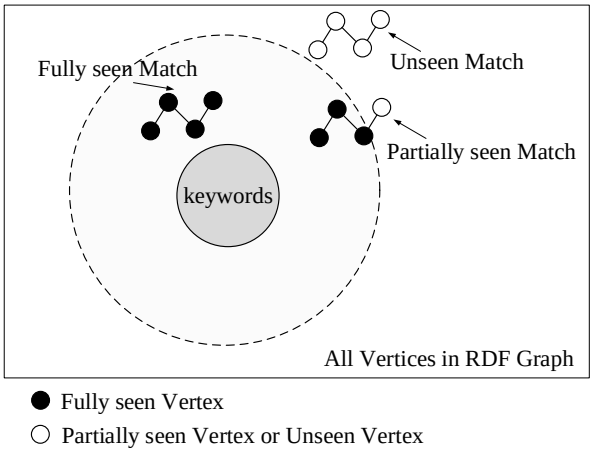


Figure 9: Fully Seen Match, Partially Seen Match and Unseen Match during the Top-k Results Computation

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 θ_1 and θ_2 for partially seen matches and unseen matches, respectively. The algorithm can stop early if and only if $\delta < \theta_1 \wedge \delta < \theta_2$. 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.

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.*

$$Cost(M) = \sum_{1 \le i \le n} d(v, w_i) \ge \sum_{d[v][w_i] \neq null \wedge 1 \le i \le n} d[v][w_i] + \sum_{d[v][w_i] = null \wedge 1 \le i \le n} |p_i|,$$

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_i.

Proof. If $d[v][i] \neq null$, it means that we have computed $d(v, w_i)$. If $d[v][w_i] = null$, v has still not been seen. Because each time we pop the head $(v, p_i, |p_i|)$ of PQ_i 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 M 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.*

$$lb(M) = \underset{v \in M}{MIN}\Big(\sum_{d[v][w_i] \neq null \wedge 1 \le i \le n} d[v][w_i] + \sum_{d[v][w_i] = null \wedge 1 \le i \le n} |p_i| \Big),$$

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

Definition 5.7. *The lower bound θ_1 for all partially seen matches is as follows.*

$$\theta_1 = MIN_{M \in PS}(lb(M))$$

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 δ and θ_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, δ is the upper bound of the top-k results, so δ 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 θ_2 for all unseen matches is as follows.*

$$\theta_2 = \underset{v \in PSet}{MIN}\Big(\sum_{d[v][w_i] \neq null \wedge 1 \leq i \leq n} d[v][w_i] + \sum_{d[v][w_i] = null \wedge 1 \leq i \leq n} |p_i| \Big),$$

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_i.

Early-stop Strategy. At each iteration step, we check whether $\delta \leq \theta_1 \wedge \delta \leq \theta_2$. 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

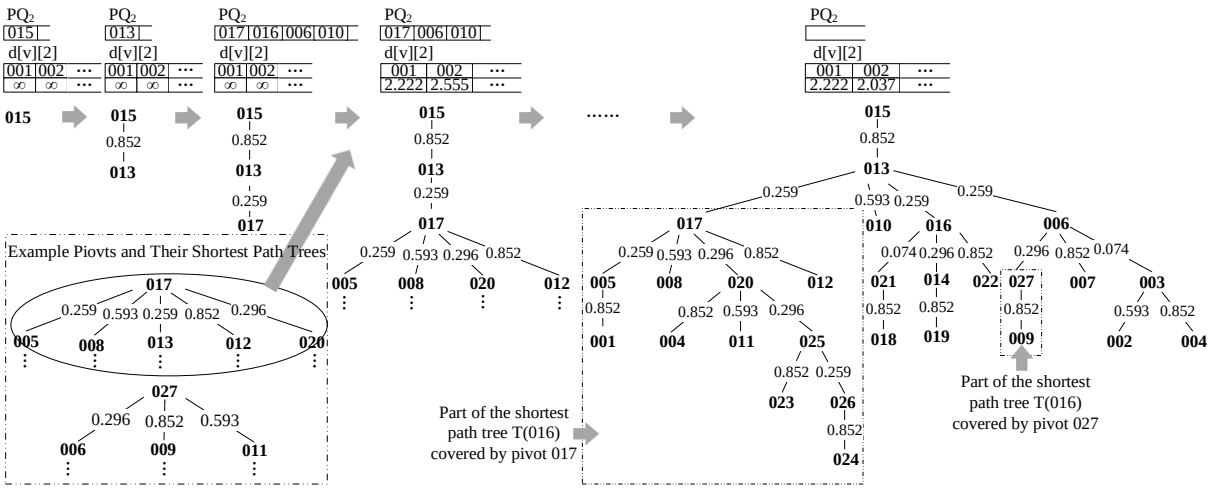


Figure 10: Pivot-based Backward Search from Keyword Vertex "Golden Globe Award for Best Actress" (Vertex "015")

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 Figure 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 Figure 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 π between r and v crosses pivot p. Hence, there are two parts in π: π_1 from r to p and π_2 from p to v. To prove that $d(r, v) = d(r, p) + d(p, v)$, we only need prove that π_1 and π_2 are the shortest paths from r to p and from p to v, respectively.

First, we prove that π_1 is the shortest path from r to p. If π_1 is not the shortest path, there exists a path π'_1 from r to p where $|\pi'_1| < |\pi_1|$. We can then create new path π' of length $|\pi'_1| + |\pi_2|$ by concatenating paths π'_1 and π_2. However, $|\pi'| = |\pi'_1| + |\pi_2| < |\pi_1| + |\pi_2| = |\pi|$. This conflicts with the fact that π is the shortest path between r and v. Therefore, π_1 is the shortest path from r to p and $d(r, p) = |\pi_1|$.

Similarly, we can prove that π_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 Figure 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_i maps to vertices in V_i. Initially, we insert all vertices into result set RS_i and all initial distances $d[v][i]$ ($\forall v \in V$) are set to $+\infty$ except for vertices in V_i, whose distances are their content costs. The differences between the pivot-based search algorithm search and the basic counterpart only happen when a queue head v popped from the queue PQ_i is a pivot (see Lines 7-17 in Algorithm 1).

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 $d[v][i] = |p_i|$. In addition, we load the shortest path tree $T(v)$. For each vertex v^* in G, if $|p_i| + d(v, v^*) < d[v][i]$, we update $d[v][i] = |p_i| + d(v, v^*)$, 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).

Algorithm 3: Pivot-based Search for Top-k Results of SK Queries

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

Output: Top-k results R of $\langle Q, q\rangle$.

```
for each vertices set V_i do
    for each vertex v in V_i do
        Insert (v, ∅, 0) into PQ_i;
while not all queues are empty and θ ≥ δ do
    for i = 1, ..., n do
        Pop the head of PQ_i (v, p, |p_i|), set d[v][i] = |p_i| and insert it into RS_i;
        if v ∈ PV then
            Update all vertices' distance to keyword w_i based on v's shortest path tree;
        else
            for each adjacent edge vv' to v do
                if p ∪ vv' is not a simple path then
                    Continue;
                if there exists another element (v', p', |p'|) in PQ_i then
                    if |p'| > |p| + ps(vv')) then
                        Replace (v', p', |p'|) with (v', p ∪ vv', |p_i| + ps(vv')) in PQ_i;
                else
                    Insert (v', p ∪ vv', |p| + ps(vv')) in PQ_i;
        if PQ_i is empty then
            for each fully seen vertex v' in PQ_i do
                Call Algorithm 2 to find all matches containing v';
                for each match M that are found do
                    if all vertices in M are fully seen vertices then
                        Use M to update R and the upper bound δ of top-k results
        Update the cost of all partially seen matches and δ;
        Update the lower bound cost θ of all remaining un-seen vertices;
Return R.
```

Example 3. *Figure 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 $d(015, 015) = 0$. 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*

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

Theorem 6.2. *Given a constant M, finding a pivot set PV to maximize $(\sum_{v \in V(G)} cr(T(v)))$ is a 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 $\{v_1, v_2, ..., v_M\}$ to maximize $\bigcup_{i=1}^{i=M} CS(v_i)$. Obviously, this is equivalent to the set cover problem and is NP-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, $Count_{update}$, 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:

$$Cost_{update} = Count_{update} \times Cost_{CPU}$$

where C_{CPU} is the average CPU cost of a distance update operation.

In addition, we suppose that $Cost_{I/O}$ 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 $Cost_{update} \leq Cost_{I/O}$, we continue to load and scan the next pivot's shortest path tree to update the tentative distances, and 2) if $Cost_{update} > Cost_{I/O}$, we end loading and scanning the next pivot's shortest path tree to update the tentative distances.

Note that both $Cost_{CPU}$ and $Cost_{I/O}$ 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 [2] 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 [35]. 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 [23]. 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 [29]. It has 19, 012, 851 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 [5]. It contains 54, 440, 096 triples. QALD [2] 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 Ghz Core 2 Duo processor and 64G RAM memory running Windows Server 2008. All experiments were implemented in Java. We used Berkeley DB[3] to store the indices. The codes and data sets are released on GitHub[4].

7.2. Effectiveness Study

In this section, we compare our methods with the classical keyword search algorithm BANKS [2] 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

[2]http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/index.php?x =challenge&q=2

[3]http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

[4]https://github.com/bnu05pp/SKQueryProcessing

the efficiency, the effectiveness of the basic search and the pivot-based search is the same.

7.2.1. Case Study

We show six sample queries in Tables 3, 4 and 5 for the case study.

	Query Sematic	SK Query	
		SPAQRL	Keywords
Q1	Which researchers on keyword search published papers in VLDB 2004 and DEXA 2005?	Select ?person where { ?paper year 2004; ?paper booktitle VLDB; ?paper1 year 2005; ?paper1 booktitle DEXA; ?paper1 creator ?person; ?paper dc:creator ?person;}	keyword search
Q2	Which papers in KDD 2005 about concept-drifting are written by Jiawei Han?	Select ?paper where { ?paper year 2003; paper booktitle KDD; ?paper creator ?person; ?person name Jiawei Han;}	concept-drifting

Table 3: Sample DBLP Queries for the Case Study

	Query Sematic	SK Query	
		SPAQRL	Keywords
Q_3	Which actors/actresses played in Philadelphia are mostly related to Academy Award and Golden Globe Award?	Select ?p where{ ?p type actor; ?p actedIn ?f; ?f label "Philadelphia"; }	Academy Award, Golden Globe Award
Q_4	Which Turing Award winners in the field of database are mostly related to Toronto?	Select ?p where { ?p type scientist; ?p hasWonPrize ?a; ?a label "Turing Award";}	Toronto, database

Table 4: Sample Yago Queries 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.

	Query Sematic	SK Query	
		SPAQRL	Keywords
Q_5	Which states of Germany are governed by the Social Democratic Party?	Select ?s where { ?s country ?g; ?g name "Germany";}	Social Democratic Party
Q_6	Which monarchs of the United Kingdom were married to a German?	Select ?u where { ?u spouse ?s; ?s birthPlace ?c; ?c name "Germany";}	United Kingdom, monarch

Table 5: Sample QALD Queries on DBPedia for the Case Study

Q_1: *Which researchers on keyword search published papers in VLDB 2004 and DEXA 2005?*

The three results returned in SK query are three researchers named "Kesheng Wu", "Jeffrey Xu Yu" and "Maurice van Keulen". All of them published papers in both VLDB 2004 and DEXA 2005. Furthermore, they wrote had written previous papers about keyword search. However, the first three results returned by the traditional keyword search are "Katsumi Tanaka", "Mong-Li Lee" and "Reda Alhajj". Although all of them are interested in keyword search, none of them had published a paper in VLDB 2004.

Q_2: *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 Q_2. 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 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.

Q_3: *Which actors/actresses played in Philadelphia are mostly related to Academy Award and Golden Globe Award?*

We had analyzed query Q_3 in Section 1. For comparison, we use the keywords {*actors*, *actresses*, *Philadelphia*, *Academy Award*, *Golden Globe Award*} for the

	Top-3 Results of SK Query	Top-3 Results of BANKS
Q_1	Kesheng Wu	Katsumi Tanaka
	Jeffrey Xu Yu	Mong-Li Lee
	Maurice van Keulen	Reda Alhajj
Q_2	Mining concept-drifting data streams using ensemble classifiers	On Reducing Classifier Granularity in Mining Concept-Drifting Data Streams.
	CLOSET+: searching for the best strategies for mining frequent closed itemsets.	ACE: Adaptive Classifiers-Ensemble System for Concept-Drifting Environments.
	CloseGraph: mining closed frequent graph patterns.	Baile Shi

Table 6: Effectiveness Results for Sample DBLP Queries

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.

Q_4: *Which Turing Award winners 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 Q_4. 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.

	Top-3 Results of SK Query	Top-3 Results of BANKS
Q_3	Denzel Washington	Grace Kelly
	Joanne Woodward	George Cukor
	Antonio Banderas	Joanne Woodward
Q_4	Stephen Cook	English language
	William Kahan	Princeton University
	Kenneth E. Iverson	Turing Award

Table 7: Effectiveness Results for Sample Yago Queries

	Top-3 Results of SK Query	Top-3 Results of BANKS	Top-3 Results of Annotated SPARQL
Q_5	Hanau	Australia	Hans-Ulrich Rudel
	Hanhofen	Bombardier Transportation	Hans Dauser
	Hanover	Canada	Hans Heidtmann
Q_6	William IV of the United Kingdom	2004 Amsterdam Admirals season	William IV of the United Kingdom
	Carl XVI Gustaf of Sweden	2004 Berlin Thunder season	Beatrix of the Netherlands
	Beatrix of the Netherlands	2004 Cologne Centurions season	Switzerland

Table 8: Effectiveness Results on DBPedia for Sample QALD Queries

Q_5: *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. However, 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

		NDCG@3	NDCG@5	NDCG@10
Yago	BANKS	0.3455	0.39	0.4643
	SK query	0.815	0.868	0.872
DBLP	BANKS	0.7143	0.684	0.685
	SK query	0.93	0.8867	0.8738

Table 9: Average NDCG Values

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) [18] 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) [32] to compare the SK query with BANKS and Annotated SPARQL.

	BANKS	Annotated SPARQL	SK query
MAP	0.012	0.192	0.205

Table 10: MAP Value over DBPedia and QALD

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

	Q_1	Q_2	Q_3	Q_4	Q_5
Exhaustive Computing	292268	21885	254674	872426	2747
Basic Search/Pivot-based Search	354	5	1684	669	1548

Table 11: Number of Graph Matching Operations on DBLP

	Q_1	Q_2	Q_3	Q_4	Q_5
Exhaustive Computing	600283	563736	301	167958	231210
Basic Search/Pivot-based Search	6	55	269	5414	32
	Q_6	Q_7	Q_8		
Exhaustive Computing	271929	94012	254848		
Basic Search/Pivot-based Search	9	292	15		

Table 12: Number of Graph Matching Operations on Yago

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.

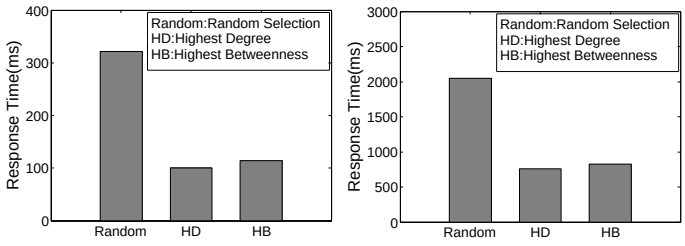
Tables 11, 12, and 13 show the number of graph matching operations on D-BLP, Yago and DBPedia, respectively. The number of graph matching operations in advanced backward search is not less than it is for the basic backward search. In most cases, we avoid a large number of graph matching operations.

	Q_1	Q_2	Q_3	Q_4	Q_5
Exhaustive Computing	31083	136777	19904	847	19454
Basic Search/Pivot-based Search	13824	3941	4769	847	40
	Q_6	Q_7	Q_8	Q_9	Q_{10}
Exhaustive Computing	40302	5076	23422	16079	2786
Basic Search/Pivot-based Search	89	18	23422	16079	1

Table 13: Number of Graph Matching Operations on DBPedia

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, highest degree-based selection (finding the M highest degree vertices as pivots), and largest betweenness-based selection (finding M largest degree of betweenness as pivots). Here, we set M to be a default value of 500. The effect of M is evaluated shortly. Figure 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 M vertices with the highest degrees than those with the largest betweenness, we use degree-based selection



(a) Results on Yago (b) Results on DBLP

Figure 11: Backward Search Performance with Different Pivot Selection Methods

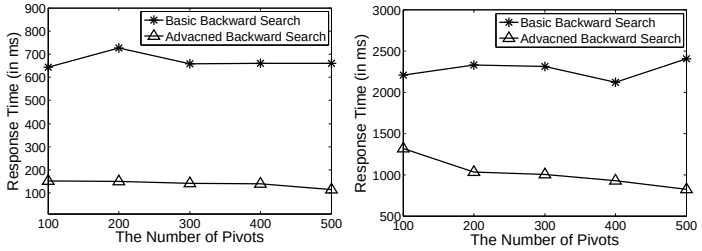
7.3.3. Evaluation of Pivot Numbers

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. Figure 12 shows that query response times decrease for all methods when the pivot number was varied from 100 to 500. Obviously, the

		Structural Index	Distance Index
DBLP	Index Construction Time(s)	77.885	1423.63
	Index Size(MB)	377.977	560
Yago	Index Construction Time(s)	176.67	3229.27
	Index Size(MB)	844.066	6260
DBPedia	Index Construction Time(s)	600.263	10676.68
	Index Size(MB)	283.559	7565

Table 14: Index Size and Index Construction Time

more pivots we choose, the smaller the query response time is. However, more pivots lead to larger space cost. Because of the space cost, we set the number of pivots to 500 in this paper.



(a) Results on Yago (b) Results on DBLP

Figure 12: Backward Search Performance with Respect to Number of Pivots

7.3.4. Offline Performance

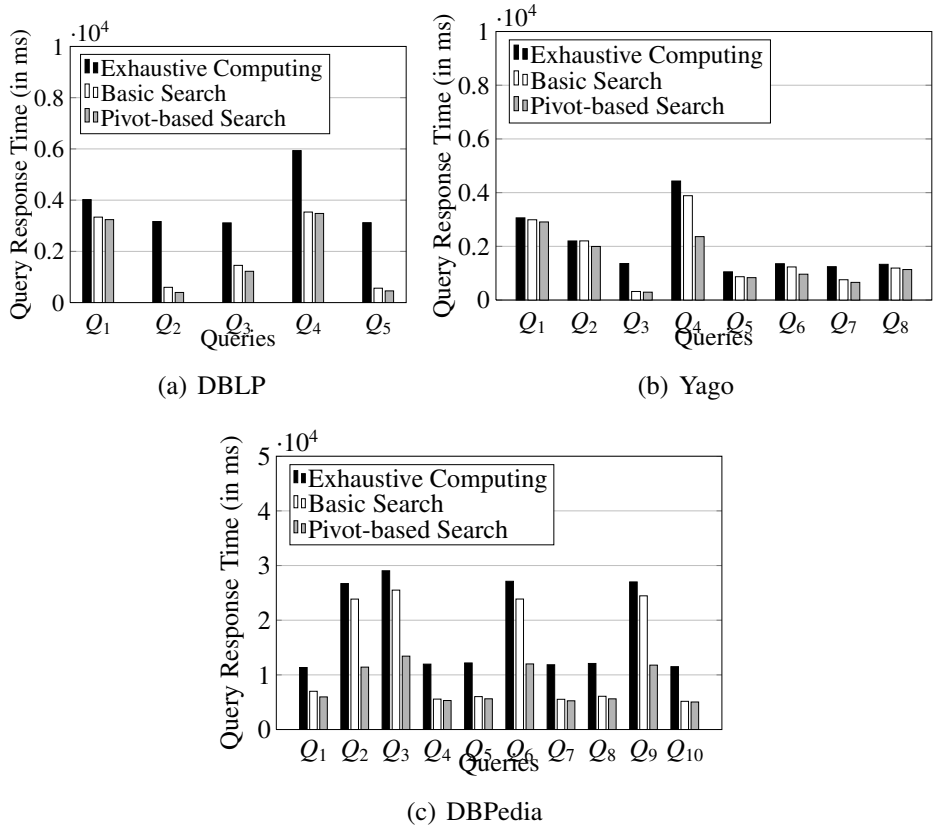
We report the index size and index construction time in Table 14. Because our structural index is based on efficient sequential pattern mining, we can finish the structural index construction in several minutes. Similarly, for our pivots-based index, we only need to compute some shortest path trees offline,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

7.3.5. Online Performance

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

As shown in Figure 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.



(a) DBLP

(b) Yago

(c) DBPedia

Figure 13: Online Performance

8. Related Work

There have been many studies to investigate the processing of SPARQL queries, such as [37, 1, 24, 25, 43, 6, 17]. Some [1, 6] store the RDF triples into an RDBM-

S and answer the SPARQL via join operations. RDF-3x [24, 25] and Hexastore [37] create indexes for each permutation of subject, predicate and object. Because an RDF dataset can also be modeled as a graph, gStore [43] and AMbER [17] 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 [8] to answer a SPARQL query. VF2 [8] 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 [36, 31, 13, 14] interpret keywords as SPARQL queries and then retrieve results by involving 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 [2, 19, 16, 11, 21, 22, 34, 22], cliques [20] or other patterns defined over the RDF graphs [9], 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 [39, 38, 40]. Among these works, gIndex [39] and gSpan [38] can be applied to small graphs in a database of multiple graphs, but do not support mining patterns in a single graph. GADDI [40] tries to finding all the matches of a query graph in a given 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 [12, 35] on hybrid queries combining SPARQL and keywords, there is no existing work on the SK query defined as above. Elbassuoni et al. [12] assumes that each RDF triple may have associated text passages. Then, they extend the triple patterns in SPARQL with keyword conditions. Moreover, CE^2 [35] 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 [28], the authors define a new query language that blends keyword search with structured query processing. [30] utilizes some given kinds of SPARQL to improve the result of object retrieval. Moreover, [3, 4] try to extend keyword search with semantics. Zou et al. [42, 41] 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.

References

[1] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach. SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. *VLDB J.*, 18(2):385–406, 2009.

[2] B. Aditya, Gaurav Bhalotia, Soumen Chakrabarti, Arvind Hulgeri, Charuta Nakhe, Parag, and S. Sudarshan. BANKS: Browsing and Keyword Searching in Relational Databases. In *VLDB*, 2002.

[3] Ravish Bhagdev, Sam Chapman, Fabio Ciravegna, Vitaveska Lanfranchi, and Daniela Petrelli. Hybrid search: Effectively combining keywords and semantic searches. In *ESWC*, pages 554–568, 2008.

[4] Nikos Bikakis, Giorgos Giannopoulos, Theodore Dalamagas, and Timos K. Sellis. Integrating keywords and semantics on document annotation and search. In *OTM Conferences*, pages 921–938, 2010.

[5] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - A Crystallization Point for the Web of Data. *Journal of Web Semantics*, 7(3):154–165, 2009.

[6] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building An Efficient RDF Store over A Relational Database. In *SIGMOD*, pages 121–132, 2013.

[7] Rudi Cilibrasi and Paul M. B. Vitányi. The google similarity distance. *IEEE Trans. Knowl. Data Eng.*, 19(3):370–383, 2007.

[8] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[9] Ananya Dass, Cem Aksoy, Aggeliki Dimitriou, Dimitri Theodoratos, and Xiaoying Wu. Diversifying the results of keyword queries on linked data. In *WISE*, pages 199–207, 2016.

[10] Dong Deng, Guoliang Li, and Jianhua Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD*, pages 673–684, 2014.

[11] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, pages 836–845, 2007.

[12] Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, and Gerhard Weikum. Searching RDF graphs with sparql and keywords. *IEEE Data Eng. Bull.*, 33(1):16–24, 2010.

[13] Haizhou Fu and Kemafor Anyanwu. Effectively interpreting keyword queries on RDF databases with a rear view. In *ISWC*, pages 193–208, 2011.

[14] Haizhou Fu, Sidan Gao, and Kemafor Anyanwu. Cosi: context-sensitive keyword query interpretation on RDF databases. In *WWW (Companion Volume)*, pages 209–212, 2011.

[15] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, 1979.

[16] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Blinks: Ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[17] Vijay Ingalalli, Dino Ienco, Pascal Poncelet, and Serena Villata. Querying RDF Data Using A Multigraph-based Approach. pages 245–256, 2016.

[18] Kalervo Järvelin and Jaana Kekäläinen. IR evaluation methods for retrieving highly relevant documents. In *SIGIR*, pages 41–48, 2000.

[19] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional Expansion For Keyword Search on Graph Databases. In *VLDB*, pages 505–516, 2005.

[20] Mehdi Kargar and Aijun An. Keyword search in graphs: Finding r-cliques. *PVLDB*, 4(10):681–692, 2011.

[21] Gjergji Kasneci, Maya Ramanath, Mauro Sozio, Fabian M. Suchanek, and Gerhard Weikum. Star: Steiner-tree approximation in relationship graphs. In *ICDE*, pages 868–879, 2009.

[22] Wangchao Le, Feifei Li, Anastasios Kementsietsidis, and Songyun Duan. Scalable Keyword Search on Large RDF Data. *IEEE Trans. Knowl. Data Eng.*, 26(11):2774–2788, 2014.

[23] Michael Ley and Patrick Reuther. Maintaining an online bibliographical database: The problem of data quality. In *EGC*, pages 5–10, 2006.

[24] Thomas Neumann and Gerhard Weikum. RDF-3X: A RISC-style Engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[25] Thomas Neumann and Gerhard Weikum. x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases. *PVLDB*, 3(1):256–263, 2010.

[26] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *ICDE*, pages 215–224, 2001.

[27] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[28] Jeffrey Pound, Ihab F. Ilyas, and Grant E. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *SIGMOD*, pages 423–434, 2010.

[29] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.

[30] Alberto Tonon, Gianluca Demartini, and Philippe Cudré-Mauroux. Combining inverted indices and structured search for ad-hoc object retrieval. In *SIGIR*, pages 125–134, 2012.

[31] Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data. In *ICDE*, pages 405–416, 2009.

[32] Andrew Turpin and Falk Scholer. User performance versus precision measures for simple search tasks. In *SIGIR*, pages 11–18, 2006.

[33] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[34] Dong Wang, Lei Zou, Wanqiong Pan, and Dongyan Zhao. Keyword graph: Answering keyword search over large graphs. In *ADMA*, pages 635–649, 2012.

[35] Haofen Wang, Thanh Tran, Chang Liu, and Linyun Fu. Lightweight integration of IR and DB for scalable hybrid search with integrated ranking support. *J. Web Sem.*, 9(4):490–503, 2011.

[36] Haofen Wang, Kang Zhang, Qiaoling Liu, Thanh Tran, and Yong Yu. Q2semantic: A lightweight keyword interface to semantic search. In *ESWC*, pages 584–598, 2008.

[37] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB*, 1(1):1008–1019, 2008.

[38] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

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

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

[41] Weiguo Zheng, Lei Zou, Xiang Lian, Jeffrey Xu Yu, Shaoxu Song, and Dongyan Zhao. How to Build Templates for RDF Question/Answering: An Uncertain Graph Similarity Join Approach. In *SIGMOD*, pages 1809–1824, 2015.

[42] Lei Zou, Ruizhe Huang, Haixun Wang, Jeffrey Xu Yu, Wenqiang He, and Dongyan Zhao. Natural language question answering over RDF: a graph data driven approach. In *SIGMOD*, pages 313–324, 2014.

[43] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gStore: Answering SPARQL Queries via Subgraph Matching. *PVLDB*, 4(8):482–493, 2011.

Appendix A. Queries in Experiments

Table A.15 and A.16 shows 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.

	Query Sematic	SK Query	
		SPAQRL	Keywords
Q1	Which researchers on keyword search published papers in VLDB 2004 and DEXA 2005?	Select ?person where { ?paper year 2004 ?paper booktitle VLDB ?paper1 year 2005 ?paper1 booktitle DEXA ?paper1 creator ?person ?paper creator ?person}	keyword search
Q2	Which papers in KDD 2005 about concept-drifting are written by Jiawei Han?	Select ?paper where { ?paper year 2003 paper booktitle KDD ?paper creator ?person ?person name Jiawei Han}	concept-drifting
Q3	Who wrote a paper in ICDM 2005 with others and knew Tamer?	Select ?person1 where { ?paper year 2005; ?paper booktitle “ICDM”; ?paper creator ?person1; ?paper dc:creator ?person2}	Tamer
Q4	Who wrote a paper VLDB 2005 and kept a good relationship to Jian Pei and Wen Jin?	Select ?person2 where { ?paper year “2005”; ?paper booktitle “VLDB”; ?paper creator ?person2;}	Jian Pei, Wen Jin
Q5	Which two researchers did research about skyline and coauthored a paper in VLDB 2005?	Select ?person1, ?person2 where { ?paper year “2005”; ?paper booktitle “VLDB”; ?paper dc:creator ?person1; ?paper dc:creator ?person2}	Skyline

Table A.15: Sample Queries over DBLP

	Query Sematic	SK Query	
		SPAQRL	Keywords
Q1	Which actresses played in Philadelphia are mostly related to Academy Award and Golden Globe Award?	Select ?p where{ ?p type actor; ?p actedIn ?f; ?f label “Philadelphia”; }	Academy Award, Golden Globe Award
Q2	Which Turing Award winners in the field of database are mostly related to Toronto?	Select ?p where { ?p type scientist; ?p hasWonPrize ?a; ?a label “Turing Award”;}	Toronto, database
Q3	Which Microsoft’s products are about SDK?	Select ?c where { ?c type company; ?c label “Microosft”; ?c created ?s;}	SDK
Q4	Which English film producers did act in a Comedy film and relate to Peking University?	Select ?p where { ?p actedIn ?f1; ?p y:created ?f2; ?f1 type ComedyFilms; ?f2 y:hasProductionLanguage English;}	Peking University
Q5	Which top members of Communist Party of China are related Kissinger?	Select ?p where { ?p isAffiliatedTo ?u; ?u label “Communist Party of China”; ?p type Politician;}	Kissinger
Q6	Whose father was United States Army generals and took part in Normandy Invasion?	Select ?p1 where { ?p hasChild ?p1; ?p type UnitedStatesArmyGenerals ;}	Normandy Invasion
Q7	Which state generated a Los Angeles Lakers player that relate to Eagle, Colorado?	Select ?p where { ?p bornIn ?c; ?c locatedIn ?s; ?p type LosAngelesLakersPlayers;}	Eagle Colorado
Q8	Which participants of People’s National Congress did graduate from universities in Beijing?	Select ?p where { ?p graduatedFrom ?u; ?u type UniversitiesInBeijing;}	People’s National Congress

Table A.16: Sample Queries over Yago

	Query Sematic	SK Query	
		SPAQRL	Keywords
Q1	Which states of Germany are governed by the Social Democratic Party?	Select ?s where { ?s country ?g; ?g name “Germany”;}	Social Democratic Party
Q2	Which monarchs of the United Kingdom were married to a German?	Select ?u where { ?u spouse ?s; ?s birthPlace ?c; ?c name “Germany”;}	United Kingdom, monarch
Q3	Which capitals in Europe were host cities of the summer olympic games?	Select ?u where { ?s type Country; ?s capital ?u;}	Olympic games, Europe
Q4	Who produced films starring Natalie Portman?	Select ?p where { ?f type Film; ?f producer ?p;}	Natalie Portman
Q5	In which films did Julia Roberts as well as Richard Gere play?	Select ?f where { ?f type Film; ?f starring ?p; ?p name “Roberts, Julia”}	Richard Gere
Q6	List all episodes of the first season of the HBO television series The Sopranos!	Select ?u where { ?s name “The Sopranos”; ?u series ?s;}	HBO, first
Q7	In which films directed by Garry Marshall was Julia Roberts starring?	Select ?f where { ?f type Film; ?f director ?p; ?p name ”Marshall, Garry”;}	Julia Roberts
Q8	Which software has been developed by organizations founded in California?	Select ?u where { ?c type Organisation; ?u developer ?c; ?u type Software;}	California
Q9	Which U.S. states possess gold minerals?	Select ?s where { ?s type Place; ?s country ?g; ?g name “the United States”;}	gold, mineral
Q10	Which countries in the European Union adopted the Euro?	Select ?u where { ?u type Country; ?u ethnicGroup European Union;}	Euro

Table A.17: Sample QALD Queries over DBPedia