Top-k Relevant Semantic Place Retrieval on Spatial RDF Data*

Jieming Shi
Department of Computer
Science
University of Hong Kong
jmshi@cs.hku.hk

Dingming Wu
College of Computer Science
& Software Engineering
Shenzhen University, China
dingming@szu.edu.cn

Nikos Mamoulis
Department of Computer
Science and Engineering
University of Ioannina
nikos@cs.uoi.gr

ABSTRACT

RDF data are traditionally accessed using structured query languages, such as SPARQL. However, this requires users to understand the language as well as the RDF schema. Keyword search on RDF data aims at relieving the user from these requirements; the user only inputs a set of keywords and the goal is to find small RDF subgraphs which contain all keywords. At the same time, popular RDF knowledge bases also include spatial semantics, which opens the road to location-based search operations. In this work, we propose and study a novel location-based keyword search query on RDF data. The objective of top-k relevant semantic places (kSP) retrieval is to find RDF subgraphs which contain the query keywords and are rooted at spatial entities close to the query location. The novelty of kSP queries is that they are location-aware and that they do not rely on the use of structured query languages. We design a basic method for the processing of kSP queries. To further accelerate kSP retrieval, two pruning approaches and a data preprocessing technique are proposed. Extensive empirical studies on two real datasets demonstrate the superior and robust performance of our proposals compared to the basic method.

1. INTRODUCTION

With the proliferation of knowledge-sharing communities like Wikipedia and the advances in automated information extraction from the Web, large knowledge bases like DBpedia [5] and YAGO [12] are constructed and made available to the public. Such knowledge bases typically adopt the Resource Description Framework (RDF) data model, which represents the data as collections of $\langle subject, predicate, object \rangle$ triples. RDF models data as entities (subjects) which are linked to other entities and/or types or descriptions (i.e., objects could be entities, types, or literals). For instance, triple $\langle Montmajour\_Abbey, dedication, Saint\_Peter \rangle$ models the fact that Montmajour Abbey is dedicated to Saint Peter. Therefore, an RDF knowledge base can also be seen as a directed graph, where nodes are entities (or types/literals) and the edges are predicates which describe the relationships between nodes.

The English version of DBpedia currently describes 4.5M entities, roughly including 1,4M persons, 883K places, 411K creative works, 241K organizations, 251K species, etc. YAGO includes more than 10M entities (like persons, organizations, cities) and contains more than 120M facts about these entities. Data.gov [4] is the largest open-government, data-sharing website that has more than a thousand datasets in RDF format with a total of 6.4 billion triples to date, covering information from business, finance, health, education, local government, etc. Many excellent applications have been developed on top of these data [32], e.g., *Hospital Compare* [6], *Patients Like Me* [9], *Alternative Fueling Station Locator* [1], *Crime in Chicagoland* [3], *SpotCrime* [10].

Keyword Search on RDF Data. RDF data are traditionally accessed with the help of a structured query language, like SPARQL [50]. However, a standard SPARQL query over RDF data requires query issuers to fully understand the language itself and be aware of the data domain. Hence, SPARQL limits data access mostly to domain experts, since it is not friendly to common users. Given this, a *keyword search* model on RDF data emerged [23, 43, 52]. This model allows users to retrieve information from RDF knowledge bases without the direct use of SPARQL-like languages and without knowledge of the RDF data domain. RDF data belongs to the category of linked data and can be modeled as a directed graph with subjects and objects as vertices and predicates as directed edges. For the purpose of keyword search, this graph can be simplified [43] by eliminating outgoing edges from subjects which connect to types or literals and by collecting all the keywords in the URIs, types, and literals of such entities to form a *unified textual description* for each vertex. A keyword query retrieves a set of (small) subgraphs where the vertices of each subgraph collectively cover all the given keywords. Specifically, each of the retrieved subgraphs includes (i) a *root* node (which is central to the subgraph), (ii) a number of *keyword* nodes, each containing one of the query keywords, and (iii) the shortest paths that connect the keyword nodes to the root. The sum of the lengths of these paths define a *looseness* score for the subgraph [26, 43, 52]. Subgraphs of low looseness are more appropriate as keyword query answers and returned, because they represent a compact and coherent part of the knowledge base related to the keywords. This, in analogy to finding the smallest (tuple) subgraphs in relational keyword query search [35] and general keyword search on graphs [31].

Example 1 Figure 1(a) shows the graph representation of several triples extracted from DBpedia. Both circles and squares are vertices in the RDF graph, representing entities. The edges (labeled by predicates) model the relationships between entities. Each entity is associated to a textual description (document) extracted

p_1: {abbey, montmajour}
v_1: {architecture, romanesque, subject}
v_2: {catholic, dedication, peter, roman, saint}
v_3: {ancient, arles, diocese}
v_4: {architectural, history, subject}
v_5: {ancient, birthplace, empire, roman}
p_2: {catholic, diocese, roman}
v_6: {mary, magdalene, patron}
v_7: {catholic, church, denomination, history}
v_8: {anatolia, ancient, deathPlace, history}

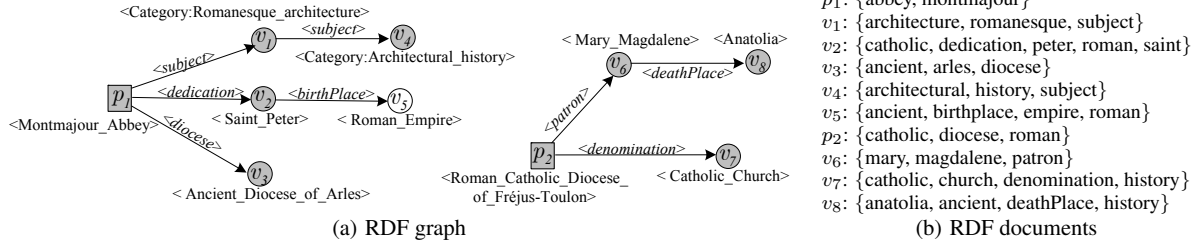(a) RDF graph   (b) RDF documents

Figure 1: RDF example

from its URI, predicates, and literals [43]. Figure 1(b) displays the documents of all vertices in Figure 1(a) (due to space constraints, for each document, only some of the terms are shown). Consider an example keyword query {$ancient, roman, catholic, history$}. According to [31, 43], the top-1 answer to the keyword query is the subgraph consisting of vertices {p_2, v_6, v_7, v_8} rooted at p_2, as well as the edges among them. This subgraph is the most compact one (i.e., it has the lowest looseness) among those whose vertices (i.e., their documents) collectively cover all query keywords. The looseness of the subgraph equals 3 and it is calculated as the sum of the lengths of the shortest paths from a common root vertex (i.e., p_2) to each matched vertex w.r.t each query keyword (i.e., p_2 for keywords {$catholic, roman$}, v_7 for keyword {$history$}, and v_8 for keyword {$ancient$}).

Spatial RDF data. Recently, RDF data have been enriched to include additional semantics. For example, YAGO2 [34] is an extension of the YAGO knowledge base that includes spatial and temporal knowledge. Enriched knowledge bases open the road to additional search and analysis operations, such as location-based retrieval. Indicatively, a key research direction of BBC News Lab is: *How might we use geolocation and linked data to increase relevance and expose the coverage of BBC News?* [2]. To fully utilize spatially enriched RDF data, the GeoSPARQL standard [14], defined by the Open Geospatial Consortium (OGC), extends RDF and SPARQL to represent geographic information and support spatial queries. RDF stores such as Virtuoso [11], Parliament [8], Strabon [42] are developed to support GeoSPARQL features. Recently, Liagouris et al. [46] extended the RDF-3X data store [48] such that the locations of spatial entities are encoded into their IDs; this facilitates efficient evaluation of spatial search operations in GeoSPARQL queries. Still, all these systems share the drawback of having to use a structured query language (SPARQL), which limits the access of common users to RDF data, as already discussed.

k**SP Queries.** In this paper we propose a novel way of searching spatial RDF data, namely the *top-k relevant Semantic Place retrieval* (kSP) query, which combines keyword search with location-based retrieval. kSP queries share the same motivation as RDF keyword queries; they are independent of the data domain and do not rely on structured languages such as SPARQL, which makes them friendly to ordinary users. They take a query location, a set of query keywords, and the number k of requested places as arguments, and they return as result the top-k *Tightest Qualified Semantic Places* (TQSP) according to a ranking function that considers both the spatial distance to the query location and the graph proximity of the occurrences of keywords in the RDF graph to the places. A *qualified semantic place* satisfies two conditions: (i) it is a tree rooted at a *place entity* (i.e., a vertex of the RDF graph associated to a spatial location, e.g., via hasGeometry predicates), (ii) the documents associated to all the vertices in the tree collectively cover all

query keywords. In accordance to existing work on RDF keyword search [23,43,52], the *looseness* score of a qualified semantic place is measured by aggregating the graph distances between the place (root) and the occurrences of the covered keywords at the nodes of the tree. The kSP query returns the k places with the smallest *combined looseness and spatial distance* to the query location, based on an aggregate function (e.g., weighted sum).

Example 2 Consider again the RDF data in Example 1 and assume that the spatial coordinates of p_1 and p_2 in Figure 1(a) are as shown in Figure 2. Assume an 1SP query issued by a tourist at location q_1 in Figure 2 who wants to do field research related to keywords {$ancient, roman, catholic, history$}. The result would be the semantic place consisting of vertices {p_1, v_1, v_2, v_3, v_4}, rooted at p_1, indicating that Montmajour Abbey (p_1), is a promising site with respect to spatial distance and semantic relevance. If the tourist was at location q_2 in Figure 2, a 1SP query with the same keywords would retrieve the semantic place consisting of vertices {p_2, v_6, v_7, v_8}, rooted at p_2, i.e., Roman Catholic Diocese.

In simple words, the objective of a kSP query is to find places that are near a given query location and they are related to a set of search keywords. Compared to RDF keyword search, the kSP query has the following unique features: (i) it retrieves *semantic places*, i.e., only subgraphs rooted at a place entity, and (ii) it is query-location-aware. kSP queries find many applications, besides the one described in Example 2. For instance, they can be used by patients who want to find nearby hospitals which offer treatment for specific conditions, companies which want to investigate the business environment of some potential nearby sites, journalists who want to search for facts related to location-dependent subjects, etc.

Figure 2: Map of places in Figure 1(a) and query points

Data Representation and Indexing. To our knowledge, this is the first work that proposes and studies kSP queries; therefore, no existing system and algorithm supports their evaluation. Typical RDF stores are designed for the efficient support of SPARQL queries, however, kSP queries require graph browsing and search opera-

tions (e.g., breadth first search). Therefore, we opt to represent the RDF data in their *native graph form* (i.e., using adjacency lists) in memory[1], as in [59]. In addition, in a preprocessing phase, we perform the following. First, we extract the document descriptions of all vertices and index them by an inverted file, which enables finding fast the vertices that contain a given keyword in their documents. Second, we store in a table, for each vertex, the document description and the spatial location (in case of a place entity), which makes it possible to directly access the keywords and location a vertex during graph browsing. Third, all place vertices are spatially indexed by an R-tree [29], which facilitates incremental nearest place retrieval from the query location.

Query Evaluation. A possible kSP query evaluation approach would be to extend the bottom-up algorithm for keyword search on graphs [31, 43]. For each query keyword t, the algorithm first determines the set of vertices whose documents contain t. From those vertices, it explores the graph by breadth first search and finds the first common vertex that all the query keywords can reach. If this common vertex is not a place vertex, the algorithm keeps running until a common place vertex is found. This vertex together with the shortest paths leading to vertices covering all keywords would form a qualified semantic place. By continuing this search it is possible to identify all TQSPs in increasing order of looseness. For each identified place, the spatial distance can be computed and the top-k TQSPs can be reported in the end. However, there is no obvious way of determining the top-k TQSPs before finding all qualified semantic places. Therefore, this method is expected to be slow; i.e., kSP queries cannot be efficiently evaluated by a straightforward extension of keyword search approaches [31, 43].

In brief, the challenges are twofold. First, not all vertices in the graph are candidate results since kSP queries look for spatial entities only. Second, the simple application of existing approaches on RDF keyword search (e.g., [31, 43]) is inefficient. As an alternative, we propose a <u>B</u>asic <u>S</u>emantic <u>P</u>lace retrieval algorithm (BSP) that retrieves the place vertices in the RDF graph in ascending order of their spatial distances to the query location using the R-tree. For each retrieved place vertex p, BSP computes the corresponding TQSP, i.e., the smallest subtree of the RDF data graph, which is rooted at p and covers all query keywords. TQSP computation is done by browsing the graph from p in a BFS manner until the query keywords are covered. The top-k places are returned as the kSP results when there is no chance for the place vertices that have not been retrieved yet (based on lower bounds of their scores) to outrank the top-k places so far.

BSP is also inefficient because it computes the TQSP of each candidate place, an expensive operation for place vertices that either cannot cover all the query keywords or have worse ranking scores than the top-k places so far. Hence, we propose two approaches for pruning the search space. The first discards *unqualified* places which do not have a TQSP covering all query keywords. The second one prunes places by aborting their TQSP computation early, based on dynamically derived bounds on their looseness. The extension of BSP which applies the two pruning techniques is referred to as <u>S</u>emantic <u>P</u>lace retrieval with <u>P</u>runing (SPP). To further improve the performance of kSP search, we introduce a data preprocessing technique, which aggregates for each place and for sets of nearby places the keywords covered by the vertices in their α-radius neighborhoods (in the RDF data graph). By indexing the preprocessed data, we can define pruning rules for place vertices and for the R-tree nodes that spatially index them. We design a

---

[1]Disk-based graph representations for RDF data (e.g., [60]) can also be used for larger-scale data.

Semantic <u>P</u>lace retrieval algorithm (SP) which applies these rules in addition to the pruning techniques of SPP. An extensive empirical study with two real data sets confirms the effectiveness and robustness of SP.

Outline. Section 2 introduces the definition of the kSP queries and relevant concepts. BSP is presented in Section 3. We present the pruning rules in Section 4 and the α-radius based bounds and related pruning techniques in Section 5. Our empirical study is reported in Section 6. Related work is reviewed in Section 7 and we conclude in Section 8.

2. PROBLEM DEFINITION

An RDF knowledge base can be modeled as a directed graph where each vertex v_i refers to an entity and edges represent triples that associate entities based on predicates. Some of the entities are associated to spatial coordinates. We call such entities *place vertices* or places for short. We use v to denote any vertex in the RDF graph, while p is especially used to denote place vertices. Each RDF triple corresponds to a directed edge from an entity (subject) to another entity (object). In accordance to previous work on RDF keyword search, we construct, for each entity, a document ψ from the entity's URI and literals. In addition, for each triple, the description of the predicate is added to the document of the object entity. A *semantic place* is a sub-tree of the RDF graph rooted at a place vertex. Given a place vertex as the root, multiple semantic places can be constructed. In other words, in the RDF graph, a place is associated to multiple semantics by being connected to different vertices.

Example 3 In Figure 1(a), the squares are place vertices and the circles are non-place vertices. Figure 1(b) shows (part of) the documents attached to the vertices. The tree consisting of vertices $\{p_1, v_1, v_2, v_3, v_4\}$ rooted at p_1 is a semantic place. The tree rooted at p_2 with vertices $\{p_2, v_6, v_7, v_8\}$ is another semantic place.

A *top-k relevant Semantic Place retrieval* (kSP) query q consists of three arguments: the query location $q.\lambda$, the query keywords $q.\psi$, and the number of requested semantic places k. A *qualified semantic place* w.r.t. a kSP query is formally defined in Definition 1. Generally speaking, the documents of the vertices in a qualified semantic place collectively cover all the query keywords.

Definition 1. **Qualified Semantic Place.** Given a kSP query q and a RDF graph $G = \langle V, E \rangle$, a qualified semantic place is a tree $T_p = \langle V', E' \rangle$ rooted at place vertex p, such that $V' \subseteq V$, $E' \subseteq E$, and $\cup_{v \in V'} v.\psi \supseteq q.\psi$.

For the ease of presentation, in the rest of the paper, a semantic place is also denoted by $\langle p, (v_1, v_2, \cdots) \rangle$, where p is the root and $(v_1, v_2, \cdots)$ includes all the other vertices. Given a kSP query, there may exist multiple semantic places with the same root p but different $(v_1, v_2, \cdots)$ sets. Following existing work on keyword search over graphs [31, 43], we define the looseness of a qualified semantic place in Definition 2.

Definition 2. **Looseness.** Given a qualified semantic place $T_p = \langle V', E' \rangle$, let $d_g(p, t_i) = \min_{v \in V' \wedge t_i \in v.\psi} d(p, v)$ be the length of the shortest path from root p to keyword $t_i \in q.\psi$, where $d(p, v)$ is the shortest path from p to v. The looseness of T_p is defined as $L(T_p) = 1 + \sum_{t_i \in q.\psi} d_g(p, t_i)$.

Looseness aggregates the proximity of the query keywords in the qualified semantic place in terms of graph distance. We add 1 to the sum of the paths from p to the nearest occurrence of each keyword for normalization purposes (as we will see later, the case

of $L(T_p) = 0$ should be avoided in our raking function for kSP results). The smaller the looseness, the more relevant the root (i.e., the place) is to the vertices that cover the query keywords. Thus, given a place vertex p as the root, we seek for the *Tightest Qualified Semantic Place* (TQSP) for the given query keywords, which is the qualified semantic place rooted at p with the smallest looseness.[2]

Example 4 Assume that the given query keywords are $q.\psi = \{ancient, roman, catholic, history\}$. Based on the RDF example shown in Figure 1, multiple qualified semantic places can be found, such as $\langle p_2, (v_6, v_8) \rangle$, $\langle p_2, (v_6, v_7, v_8) \rangle$, and $\langle p_1, (v_1, v_2, v_3, v_4) \rangle$. The looseness of $\langle p_1, (v_1, v_2, v_3, v_4) \rangle$ is calculated by $1 + 1 + 1 + 2 + 1 = 6$, where $d_g(p_1, ancient) = 1, d_g(p_1, roman) = 1, d_g(p_1, catholic) = 1, d_g(p_1, history) = 2$. There are two qualified semantic places both rooted at p_2, but with different looseness, i.e., 5 for $\langle p_2, (v_6, v_8) \rangle$ and 4 for $\langle p_2, (v_6, v_7, v_8) \rangle$. Thus, the TQSP rooted at p_2 is $\langle p_2, (v_6, v_7, v_8) \rangle$.

Definition 3. **Top-k Relevant Semantic Place Retrieval.** Given a kSP query q on an RDF graph, the result of q includes k TQSPs minimizing ranking function $f(L(T_p), S(q, p))$, where $S(q, p)$ is the spatial distance between the query location and the root of the semantic place. Recall that each place p has a unique TQSP, therefore it may appear at most once in a kSP result.

The kSP query aims at finding the semantic places that (i) are spatially close to the query location, (ii) cover the query keywords, and (iii) have a tree in which the query keywords are closely connected. Without loss of generality, Euclidean distance $S(q, p)$ is used as spatial distance in this work. Ranking function $f(L(T_p), S(q, p))$ can be any monotonic aggregate function which considers both $L(T_p)$ and $S(q, p)$, such as:

$$f(L(T_p), S(q, p)) = \beta \times L(T_p) + (1 - \beta) \times S(q, p), \quad (1)$$

$$f(L(T_p), S(q, p)) = L(T_p) \times S(q, p). \quad (2)$$

The kSP evaluation approaches proposed in this paper are independent of how f is defined. In the rest of the paper, we use Equation (2) as the ranking function, because it is parameterless.

Example 5 Consider an example kSP query q with query location $q.\lambda = q_1$ as shown in Figure 2 and query keywords $q.\psi = \{ancient, roman, catholic, history\}$. Places p_1 and p_2 are located at $(43.71, 4.66)$ and $(43.13, 5.97)$, respectively in Figure 2. Based on the RDF graph in Figure 1(a) and the documents in Figure 1(b), place p_1 has $S(q_1, p_1) = 0.22$ and $L(T_{p_1}) = 1+1+1+2+1 = 6$. $f(L(T_{p_1}), S(q_1, p_1)) = L(T_{p_1}) \times S(q_1, p_1) = 1.32$. Place p_2 has $S(q_1, p_2) = 1.28$ and $L(T_{p_2}) = 0 + 0 + 1 + 2 + 1 = 4$. $f(L(T_{p_2}), S(q_1, p_2)) = 5.12$. Therefore, p_1 is returned as top-1 and p_2 ranks second for the kSP query q with $q.\lambda = q_1$ and $q.\psi = \{ancient, roman, catholic, history\}$.

If the query location of the kSP query q is changed to $q.\lambda = q_2$ and the query keywords are unchanged, then $S(q_2, p_1) = 1.35$, $L(T_{p_1}) = 6$, and $f(L(T_{p_1}), S(q_2, p_1)) = 8.10$. $S(q_2, p_2) = 0.08$, $L(T_{p_2}) = 4$ and $f(L(T_{p_1}), S(q_2, p_1)) = 0.32$. For the kSP query q with $q.\lambda = q_2$ and $q.\psi = \{ancient, roman, catholic, history\}$, p_2 is returned as the top-1 SP and p_1 ranks second.

---

[2]If multiple trees rooted at p have the same minimum looseness, we can: (1) break ties arbitrarily and select one of them to be the TQSP for p or (2) keep all trees with the same minimum looseness in a set. If we use option (2), the result of a kSP query would the top-k qualified semantic place sets. The methods proposed in this paper are applicable for both options. For the ease of presentation, we adopt option (1) in the rest of the paper.

3. BASIC METHOD: BSP

The most relevant existing work to our kSP queries are the top-k keyword queries on graphs [31,43]. Given a set of query keywords, the objective is to retrieve the top-k sub-trees of the RDF graph, such that the vertices of each tree collectively cover the query keywords, ranked by the looseness of the trees. A bottom-up algorithm is used to evaluate top-k keyword queries. For each query keyword t, the algorithm first determines the set of vertices whose documents contain t. From those vertices, it starts to explore the graph and finds the earliest common vertex that all the query keywords can reach. This way, candidate trees are found and the result is finalized by choosing the top-k less-looseness trees. However, this approach is not appropriate for our kSP queries. Firstly, we aim for semantic places that take place vertices as roots, however, the aforementioned algorithm cannot guarantee that the discovered trees are rooted at place vertices. Secondly, the ranking score of a semantic place depends on both its looseness and its spatial distance to the query location; even if a keyword query can identify candidate trees rooted at places, there is no obvious way of determining the top-k semantic places before getting all candidate trees, since a tree T_p with high $L(T_p)$ value but small spatial distance $S(p, q)$ to the query location q may outrank a tree $T_{p'}$ with low $L(T_{p'})$ but large spatial distance $S(p', q)$, and vice versa.

Obviously, a single TQSP computation is much more expensive compared to a single spatial distance computation. Therefore, using keyword query (keyword-first) methods to solve our problem would be inefficient. In view of this, we design methods that perform spatial search first, in order to avoid unnecessary TQSP computations. In this section, we propose a basic method for evaluating kSP queries. This method requires that we have preprocessed the RDF graph, extracted the places from it and spatially indexed them using an R-tree [29]. Like keyword search approaches, we also assume that the documents of the vertices in RDF graph are indexed by an inverted index [37]. Table 1 shows the inverted index for the documents in Figure 1(b). In addition, instead of storing and indexing the RDF data in a triples table format, which would enable efficient SPARQL query evaluation, we choose to store the RDF graph in memory in its native form (i.e., using adjacency lists, as in [59]), which enables efficient graph browsing operations (like BFS). Finally, we keep a table which helps to look-up fast the associated data (document, spatial coordinates) for each vertex, so there is no need to employ any special encoding scheme [46].

abbey: p_1	church: v_7	magdalene: v_6
anatolia: v_8	deathPlace: v_8	montmajour: p_1
ancient: v_3, v_5, v_8	dedication: v_2	patron: v_6
architectural: v_4	denomination: v_7	peter: v_7
architecture: v_1	diocese: v_3, p_2	roman: v_2, v_5, p_2
arles: v_3	empire: v_5	romanesque: v_1
birthplace: v_5	history: v_4, v_7, v_8	saint: v_2
catholic: v_2, p_2, v_7	mary: v_6	subject: v_1, v_4

Table 1: Inverted index of the documents in Figure 1(b)

Algorithm 1 shows the pseudo code of our Basic Semantic Place (BSP) search method for evaluating kSP queries. Initially, a top-k result queue H_k, which prioritizes identified semantic places by their ranking scores, is initialized (line 1). Given a kSP query q, the posting lists of the query keywords are loaded (lines 2–3). Then, the basic method applies the best-first search algorithm [33] on the R-tree to retrieve places in ascending order of their spatial distances to the query location (line 6). For each retrieved place p from the R-tree, BSP constructs the TQSP T_p rooted at p using function GETSEMANTICPLACE() (line 9). Then, T_p is inserted into the result queue H_k (line 13). A threshold θ is set as the ranking score

of the kth semantic place in the result queue (line 14). For the next retrieved entry e from the R-tree (e may refer to a place or a node in the R-tree), if its minimum spatial distance to the query location is not smaller than the threshold, i.e., $S(q, e) \geq \theta$, the top-k result is finalized and the algorithm terminates (lines 7 and 15).

Correctness of Termination. For the next retrieved entry e, if $S(q, e) \geq \theta$, the spatial distances of all unprocessed places to the query location are not smaller than the threshold. This means that the ranking scores of all these places cannot be better than the current k^{th} candidate, given the fact that since $L(T_p) \geq 1$, we have $f(L(T_p), S(q, p)) \geq S(q, p)$. Hence, the current k candidates are correctly returned as the top-k TQSPs for q. Note that the termination condition is based on Equation 2; it can be easily adjusted if $f(L(T_p), S(q, p))$ is defined differently (e.g., using Equation 1).

Algorithm 1 BSP(q, R, G, I)

1: MinHeap $H_k = \varnothing$, ordered by $f(L(T_p), S(q, p))$
2: **for each** keyword t_i in $q.\psi$ **do**
3:      Load posting list pl_i of t_i from I
4: Construct $M_{q.\psi}$
5: $\theta = +\infty$
6: **while** e=GETNEXT(R, q) **do**
7:      **if** $S(q, e) \geq \theta$ **then break**
8:      **if** e refers to a place p **then**
9:          T_p = GETSEMANTICPLACE$(q.\psi, p, G, M_{q.\psi})$
10:          **if** $L(T_p) == +\infty$ **then continue**
11:          Compute the ranking score f of T_p
12:          **if** $f < \theta$ **then**
13:              H_k.add(T_p, f)
14:              Update θ
15: **return** H_k

Example 6 Consider a 1SP query q located at $q.\lambda = q_1$ in Figure 2 with query keywords $q.\psi = \{$*ancient, roman, catholic, history*$\}$, applying on the RDF graph of Figure 1. Place p_1 is firstly retrieved from the R-tree and $L(T_{p_1}) = 6$ after calling function GETSEMANTICPLACE(). Therefore, the ranking score of T_{p_1} is $f = 1.32$. Next, T_{p_1} is added into H_k as the top-1 candidate and θ is updated to 1.32. Similarly, place p_2 is retrieved from the R-tree with $S(q_1, p_2) = 1.28 < \theta$. The TQSP rooted at p_2 is constructed with $L(T_{p_2}) = 4$ and its ranking score is $f = 5.12$. Finally, T_{p_1}, which has a smaller score, is returned as the top-1 result.

Before calling GETSEMANTICPLACE(), for the sake of efficiency, the loaded posting lists of the query keywords are converted into a map structure $M_{q.\psi}$ where keys are the vertices in these posting lists. For each key (vertex), its value is the set of query words appeared in the document of the vertex. Taking query keywords $q.\psi = \{ancient, roman, catholic, history\}$ as an example, according to the inverted index in Table 1, the content of $M_{q.\psi}$ is shown in Table 2. Usually the number of query keywords is small, therefore, $M_{q.\psi}$ is small and cheap to construct.

v_2: {catholic, roman}
v_3: {ancient}
v_4: {history}
v_5: {ancient, roman}
v_7: {catholic, history}
v_8: {ancient, history}
p_2: {catholic, roman}

Table 2: $M_{q.\psi}$ of the example in Figure 1

Function GETSEMANTICPLACE() constructs the TQSP T_p rooted at a place p w.r.t. query q. According to the definition of TQSP, T_p contains the shortest path from the root place to each query keyword. A naive way is to compute the shortest path from

p to every vertex in the posting list pl_i of keyword t_i, and then choose the vertex with the smallest shortest path distance. For example, in Figure 1, in order to determine the shortest path from p_1 to keyword *ancient*, we need to compute the shortest path for pairs (p_1, v_3), (p_1, v_5) and (p_1, v_8), and then get $\langle p_1, v_3 \rangle$ as the shortest path from p_1 to keyword *ancient*. Apparently, when the RDF data graph is large, this approach would be expensive as it would require the computation of numerous and long shortest paths.

Instead, function GETSEMANTICPLACE() applies breadth first search (BFS) in the RDF graph, starting from the root place p, and checks whether each encountered vertex v contains any query keyword t using map $M_{q.\psi}$. Meanwhile, a keyword set B is maintained to record the undiscovered keywords during BFS. Algorithm 2 shows the pseudocode. TQSP T_p is initialized as empty (line 1), looseness $L(T_p)$ is set to 1 (line 2), and set B contains all the query keywords (line 3). BFS search starting from place p incrementally reports the next encounter vertex v (line 4). The query keywords $v.\psi_q$ associated with v can be obtained from $M_{q.\psi}$ (line 6). If B and $v.\psi_q$ share words, this means that the shortest paths from the root to some keywords in B have been identified (line 7). Then, T_p and its looseness are updated (lines 8) and these keywords are removed from B (line 9). If no more vertices are identified by BFS and B is not empty, there is no qualified semantic place rooted at p (line 10). As soon as B is empty (i.e., all query keywords have been covered), T_p is successfully constructed and returned.

Algorithm 2 GETSEMANTICPLACE$(q.\psi, p, G, M_{q.\psi})$

1: $T_p = \varnothing$
2: $L(T_p) = 1$
3: Set $B = q.\psi$
4: **while** $v = $ BFS(G, p) and $B \neq \varnothing$ **do**
5:      Add v to T_p
6:      $v.\psi_q = M_{q.\psi}$.get(v)
7:      **if** $B \cap v.\psi_q \neq \varnothing$ **then**
8:          $L(T_p)+ = |B \cap v.\psi_q| \times d(p, v)$
9:          $B = B \smallsetminus v.\psi_q$
10: **if** $B \neq \varnothing$ **then** $L(T_p) = +\infty$ and $T_p = $ NULL
11: **return** $L(T_p)$ and T_p

Example 7 Given query keywords $q.\psi = \{$*ancient, roman, catholic, history*$\}$, we illustrate function GETSEMANTICPLACE() (Algorithm 2) by constructing the TQSP for place p_1 in Figure 1. BFS firstly reports p_1 that is added to T_{p_1}. However, $B \cap p_1.\psi_q = \varnothing$, which means that there is nothing to do for p_1. Next, v_1 is visited by BFS, which again contains no keywords from B. When v_2 is visited, we have $B \cap v_2.\psi_q = \{catholic, roman\}$. Therefore, $L(T_{p_1})+ = 2 \cdot d(p, v_2) = 3$, and *catholic* and *roman* are removed from B. Similarly, after v_3 and v_4 have been processed, $L(T_{p_1})$ is updated to 6 and B becomes empty. Thus, $T_{p_1} = \langle p_1, (v_1, v_2, v_3, v_4) \rangle$ and $L(T_{p_1}) = 6$ are returned.

4. IMPROVED PRUNING: SPP

In the basic method, for each retrieved place p from the R-tree, function GETSEMANTICPLACE() is called to construct the TQSP T_p rooted at p. The effort of a TQSP construction is wasted under two circumstances: (i) T_p cannot cover all the query keywords, i.e., no qualified semantic place rooted at p can be obtained and (ii) the ranking score of T_p is no less than threshold θ (the ranking score of the kth candidate). For case (i), we design a reachability-based pruning rule that discards the places whose TQSP cannot be constructed. Using this rule, some places are pruned without calling function GETSEMANTICPLACE(). For case (ii), we derive a dynamic bound on the looseness of the TQSP under construction.

This bound is used to judge whether the unfinished TQSP has the potential to belong to the top-k result. This rule helps reducing the TQSP construction cost for some places that cannot enter the kSP result. Applying the two pruning techniques, we design a <u>S</u>emantic <u>P</u>lace retrieval with <u>P</u>runing algorithm (SPP).

4.1 Unqualified Place Pruning

A place p retrieved by the GETNEXT function of the basic algorithm may not form a qualified semantic place. This happens if it is not possible to reach vertices covering all query keywords by BFS from p. For example, consider place p_2 in Figure 1 and query keywords $\{church, architecture\}$; no qualified semantic place rooted at p_2 exists, since p_2 never reaches $architecture$. Formally:

PRUNING RULE 1. ***Unqualified Place Pruning.*** *Let $p \nrightarrow t$ denote that place p cannot reach keyword t in the RDF graph. Given query keywords $q.\psi$, place p is an unqualified place and can be pruned if $\exists t \in q.\psi, p \nrightarrow t$.*

Testing whether p can reach a keyword t in the graph can be implemented by reachability queries [20,39,40,53,58] that have been well studied in the literature. TF-Label [20] is the state-of-the-art algorithm for reachability queries, using which we can perform 1M reachability queries in a large graph within dozens of milliseconds. We use TF-Label as a independent component in our algorithm.

In the RDF graph, the documents of multiple vertices may share the same keyword t (as many as the length of the corresponding inverted list). For instance, in Figure 1(b), the documents of v_3, v_5, and v_8 all contain keyword $ancient$. Thus, in order to determine whether a place p can reach keyword $ancient$, in the worst case three reachability queries (i.e., to v_3, v_5, and v_8) have to be issued. In a very large data set, a huge number of reachability queries may have to be performed, which is inefficient. To reduce the number of reachability queries, we propose the following method. Firstly, a vertex v_t is constructed for each word t and added into the RDF graph. Edges are added from the vertices whose documents contain t to v_t. This way, for each query keyword t, it suffices to apply a single reachability query to v_t in order to find out whether any of the vertices whose documents contain t is reachable from the place vertex. Therefore, the number of required reachability queries for a place becomes at most equal to the number of query keywords. Secondly, based on the observation that infrequent query keywords have a high chance to make a place unqualified, we prioritize them when issuing reachability queries.

Pruning Rule 1 is used before calling function GETSEMANTICPLACE() (line 9 in Algorithm 1) to avoid unnecessary TQSP computations.

4.2 Dynamic Bound based Pruning

Function GETSEMANTICPLACE() constructs the TQSP T_p rooted at p in a BFS manner: starting from p its neighboring nodes are incrementally explored. During this process, some of the query keywords may be found early, while it may take time to find others. We derive a dynamic bound for the looseness of the TQSP T_p under construction in Lemma 1. This dynamic bound converges to the real looseness of TQSP as more keywords are covered.

LEMMA 1. ***Dynamic Bound on Looseness.*** *Given query keywords $q.\psi = \{t_1, ..., t_j, .., t_m\}$, without loss of generality, suppose that we have already discovered the first j query keywords during the BFS exploration starting from p. Let v be the next vertex encountered in the BFS process with graph distance $d(p,v)$. A lower bound of the looseness $L(T_p)$ is then $L_B(T_p) = \sum_{i=1}^{j} d_g(p,t_i) + d(p,v) \times (m-j)$.*

PROOF. Trivial due to the monotonicity of $L(T_p)$ w.r.t. the shortest paths to the first encounters of keywords. Vertex v is the next encountered vertex in the BFS process. Hence, all the undiscovered keywords cannot have a shorter graph distance from p than v does, i.e., $d_g(p,t_n) \geq d(p,v), j < n \leq m$. Therefore, we can have $L_B(T_p) = \sum_{i=1}^{j} d_g(p,t_i) + d(p,v) \times (m-j) \leq \sum_{i=1}^{m} d_g(p,t_i) = L(T_p)$. □

A TQSP T_p has a chance to be in the kSP result only if its ranking score is less than threshold θ. Definition 4 presents the looseness threshold for all TQSPs that have not been computed yet.

Definition 4. **Looseness Threshold.** Let θ be the ranking score of the kth TQSP found so far. The looseness threshold of any TQSP T_p is defined as $L_w(T_p) = \theta/S(q,p)$. If a TQSP has looseness no smaller than its $L_w(T_p)$, it cannot be in the kSP result.

Based on Lemma 1 and Definition 4, we introduce the dynamic bound based pruning rule in Pruning Rule 2.

PRUNING RULE 2. ***Dynamic Bound based Pruning.*** *For place p, as soon as $L_B(T_p) \geq L_w(T_p)$, the TQSP rooted at place p cannot be in the kSP result, and thus p can be pruned.*

PROOF. For the TQSP T_p rooted at p, if its best possible looseness $L_B(T_p)$ is no smaller than its looseness threshold $L_w(T_p)$, meaning that the ranking score of T_p must be no smaller than the current k^{th} candidate, then T_p cannot be in the result. □

By applying the two pruning rules, we can design algorithm SPP (<u>S</u>emantic <u>P</u>lace search with <u>P</u>runing), which is an extension of BSP. Algorithm 3 shows an improved version of function GETSEMANTICPLACE() (Algorithm 2) used in SPP. Algorithm 3 differs from Algorithm 2 in line 4 that computes the looseness threshold of the T_p to be constructed, line 7 that computes the dynamic bound on the looseness of T_p each time when BFS reports new vertex, and lines 8–9 that apply Pruning Rule 2 to prune places. Having Pruning Rules 1 and 2, SPP is Algorithm 1 with the following change. The looseness threshold in Definition 4 and Pruning Rule 2 guarantee that any place survived to the point when to be added to H_k must be ranked at least the kth position. Therefore, the *if* clause at line 12 of Algorithm 1 is not needed anymore.

Algorithm 3 GETSEMANTICPLACEP($q.\psi, p, G, M_{q.\psi}$)

1: $T_p = \varnothing$
2: $L_B(T_p) = 1$
3: Set $B = q.\psi$
4: Compute the looseness threshold $L_w(T_p)$
5: **while** $v = \text{BFS}(G,p)$ and $B \neq \varnothing$ **do**
6:     Add v to T_p
7:     Compute the dynamic bound $L_B(T_p)$
8:     **if** $L_B(T_p) \geq L_w(T_p)$ **then**   ▷ Pruning Rule 2
9:         return $+\infty$ and $T_p = \text{NULL}$
10:     $v.\psi = M_{q.\psi}.\text{get}(v)$
11:     **if** $B \cap v.\psi \neq \varnothing$ **then**
12:         $B = B \smallsetminus v.\psi$
13: **if** $B \neq \varnothing$ **then** $L(T_p) = +\infty$ and $T_p = \text{NULL}$
14: **return** $L_B(T_p)$ and T_p

Example 8 Consider a kSP query q located at $q.\lambda = q_1$ in Figure 2 with keywords $q.\psi = \{ancient, roman, catholic, history\}$, requesting the top-1 TQSP in the RDF graph of Figure 1. Place p_1 is firstly retrieved from the R-tree. After applying Pruning Rule 1, we find that p_1 can reach all the query keywords and cannot be pruned. Then TQSP T_{p_1} rooted at p_1 is constructed and

regarded as the top-1 candidate with ranking score 1.32. Threshold θ is set to 1.32. Next, place p_2 is retrieved from the R-tree, which again cannot be eliminated by Pruning Rule 1. Then, function GETSEMANTICPLACEP() is called to construct TQSP T_{p_2} rooted at p_2. The looseness threshold for T_{p_2} is calculated as $L_w(T_{p_2}) = \theta / S(q_1, p_2) = 1.32/1.28 = 1.03$. BFS starts to explore the graph starting from p_2; in the meanwhile, the dynamic bound on the looseness of T_{p_2} is computed. Initially, $L_B(T_{p_2})=1$. After p_2 is visited by BFS, $d(p_2, p_2) = 0$. By Lemma 1, $L_B(T_{p_2}) = 1 + d(p_2, p_2) \times |B| = 1$. Since $L_B(T_{p_2}) = 1 < L_w(T_{p_2})$, p_2 cannot be eliminated by Pruning Rule 2. Then, $\{catholic, roman\}$ are removed from B since they are contained in p_2 itself. After v_6 is visited by BFS, $d(p_2, v_6) = 1$, and therefore, by Lemma 1, $L_B(T_{p_2})$ is increased by $|B| \times d(p_2, v_6)$ and becomes 3. According to Pruning Rule 2, $L_B(T_{p_2}) = 3 > 1.03 = L_w(T_{p_2})$ and T_{p_2} cannot be the top-1 result. Hence, function GETSEMANTICPLACEP() returns NULL before finishing the construction of T_{p_2} and is more efficient than function GETSEMANTICPLACE() in Algorithm 2.

5. α-RADIUS BASED BOUNDS

The pruning rules proposed in the previous section help discarding unqualified places and the places whose TQSPs cannot enter the kSP result. In this section, we propose new bounds on both the looseness and the ranking scores, for pruning not only individual places but also sets of places, i.e., R-tree entries and the corresponding sub-trees. We firstly introduce the α-radius word neighborhood in Definition 5, which is used for deriving the bounds.

Definition 5. α-radius word neighborhood of place. For place p, its α-radius word neighborhood $WN(p)$ contains the set of word-distance pairs $\{(t_i, d_g(p, t_i))\}$ where the shortest graph distance from p to each word t_i is no larger than α, i.e., $d_g(p, t_i) \le \alpha$.

Based on the α-radius word neighborhood of individual places, we define the α-radius word neighborhoods of a set of places, i.e., a node in the R-tree, in Definition 6.

Definition 6. α-radius word neighborhood of node. For a set of places $\{p_j\}$ enclosed in a node N of the R-tree, the α-radius word neighborhood $WN(N)$ of N contains the set of word-distance pairs $\{(t_i, d_g(N, t_i))\}$ where the words in $WN(N)$ is the union of the words in $WN(p_j)$ of all places enclosed in N, and for each word t_i, $d_g(N, t_i) = \min_{p_j \in N} d_g(p_j, t_i)$. Obviously, $d_g(N, t_i) \le \alpha$.

Construction of α-radius word neighborhood. In a preprocessing phase, the α-radius word neighborhoods of all places are computed first. For each place p, we explore the RDF graph in a breadth-first manner starting from p. Neighborhood $WN(p)$ is initialized as empty. When encountering a vertex v in the graph, for each word t appearing in v's document, if no corresponding pair for t is already in $WN(p)$, a new pair $(t, d(p, v))$ is added to $WN(p)$. After the α-radius word neighborhoods of all places have been constructed, the α-radius word neighborhoods of the nodes in the R-tree are computed in a bottom-up fashion from the leaf level to the root level. For each node N, let $\{e_i\}$ be the set of entries enclosed, where e_i refers to either a place or a node. Neighborhood $WN(N)$ is initialized as empty. For each pair $(t, d_g(e_i, t))$ in each $WN(e_i)$, if no corresponding pair for t is in $WN(N)$, $(t, d_g(e_i, t))$ is added to $WN(N)$ as $(t, d_g(N, t))$; otherwise, $d_g(N, t)$ is updated as $\min\{d_g(N, t), d_g(e_i, t)\}$.

Example 9 For $\alpha = 1$, part of the α-radius word neighborhoods of places p_1 and p_2 in Figure 1(a) are displayed in the first two rows of Table 3. '-' indicates that the place cannot reach the keyword

within α-radius. Assuming that an R-tree node N contains p_1 and p_2, the α-radius $WN(N)$ is shown in the last row of the table.

$q.\psi$	abbey	...	ancient	catholic	roman	history	...
$d_g(p_1, t_i)$	0	...	1	1	1	-	...
$d_g(p_2, t_i)$	-	...	-	0	0	1	...
$d_g(N, t_i)$	0	...	1	0	0	1	...

Table 3: Example: 1-radius word neighborhoods

Based on the α-radius word neighborhoods of places, we derive bounds of the looseness and the ranking scores of TQSPs based on Lemmas 2 and 3. Lemmas 4 and 5 extend these bounds for sets of places rooted under R-tree nodes.

LEMMA 2. α*-bound on the looseness of a place.* Let $WN(p)$ be the α-radius word neighborhood of place p. Given query keywords $q.\psi = \{t_1, \ldots, t_j, \ldots, t_m\}$, without loss of generality, assume that the first j keywords have corresponding pairs in $WN(p)$. The α-bound of the looseness of TQSP T_p rooted at p is $L_B^\alpha(T_p) = \sum_{i=1}^{j} d_g(p, t_i) + (\alpha + 1) \times (m - j)$ and $L_B^\alpha(T_p) \le L(T_p)$.

LEMMA 3. α*-bound on the ranking score for places.* Let $L_B^\alpha(T_p)$ be the α-bound on the looseness of the TQSP T_p rooted at p. Given a kSP query q, the α-bound on the ranking score of T_p is $f_B^\alpha(p) = L_B^\alpha(T_p) \times S(q, p)$ and $f_B^\alpha(p) \le f(L(T_p), S(q, p))$.

LEMMA 4. α*-bound on the looseness for nodes.* Let $WN(N)$ be the α-radius word neighborhood of node N. Given query keywords $q.\psi = \{t_1, \ldots, t_j, \ldots, t_m\}$, without loss of generality, assume that the first j keywords have corresponding pairs in $WN(N)$. The α-bound on the looseness of all the TQSPs T_p rooted at p enclosed in N is $L_B^\alpha(T_N) = \sum_{i=1}^{j} d_g(N, t_i) + (\alpha + 1) \times (m - j)$ and $\forall p_i \in N, L_B^\alpha(T_N) \le L(T_{p_i})$.

LEMMA 5. α*-bound on the ranking score for nodes.* Let $L_B^\alpha(T_N)$ be the α-bound on the looseness of the TQSPs T_p rooted at places p enclosed in N. Given a kSP query q, the α-bound on the ranking score of all the T_p rooted at p enclosed in N is $f_B^\alpha(N) = L_B^\alpha(T_N) \times S(q, N)$, where $S(q, N)$ is the minimum spatial distance between q and N. $\forall p_i \in N, f_B^\alpha(N) \le f(L(T_{p_i}), S(q, p_i))$.

The proofs of Lemmas 2, 3, 4, and 5 are omitted for the interest of space. We proceed to introduce a pruning rule for places using Lemma 3 and a pruning rule for nodes using Lemma 5.

PRUNING RULE 3. **Place pruning.** *Given a kSP query q, let θ be the ranking score of the kth candidate TQSP and $f_B^\alpha(p)$ be the α-bound on the ranking score of the TQSP T_p rooted at p. If $f_B^\alpha(p) \ge \theta$, T_p cannot be the kSP result and p is pruned.*

PRUNING RULE 4. **R-tree node pruning.** *Given a kSP query q, let θ be the ranking score of the kth candidate TQSP and $f_B^\alpha(N)$ be the α-bound on the ranking score of the TQSPs T_p rooted at places p enclosed in N. If $f_B^\alpha(N) \ge \theta$, the TQSP rooted at any place enclosed in N cannot be the result and N is pruned.*

Example 10 Consider an R-tree node N formed by places p_1 and p_2 in Figure 1(a). For query keywords $q.\psi = \{ancient, roman, catholic, history\}$, based on Table 3 and Lemma 4, $L_B^\alpha(T_N) = 1 + 0 + 0 + 1 + 1 = 3$. Assuming the minimum spatial distance from N to a query location $q.\lambda$ is 2, by Lemma 5, $f_B^\alpha(N) = 6$. If $\theta = 5$, according to Pruning Rule 4, $f_B^\alpha(N) > \theta$ which means all the places under N, i.e., p_1 and p_2 in this example, can be pruned.

Storage. The α-radius word neighborhoods of places and nodes can be modeled as vectors. They are indexed by an inverted file.

For a kSP query, part of the neighborhoods relevant to the query keywords, i.e., the posting lists of the query keywords, are loaded in the beginning query processing, to facilitate the computation of α-based bounds and the application of Pruning Rules 3 and 4.

Algorithm. By integrating the α-radius based bounds with the SPP algorithm, we design a <u>S</u>emantic <u>P</u>lace retrieval algorithm (SP) for the processing of kSP queries, as described by Algorithm 4. SP has the following differences compared to SPP: (i) entries (referring to places and nodes) in the R-tree are processed in ascending order of their α-bounds on the ranking score rather than their spatial distance to the query location (lines 8 and 22), (ii) Pruning Rules 3 and 4 are used to discard places having no potential to be the result (line 21), and (iii) the termination condition is based on the α-bound on the ranking score rather than the spatial distance, which can be satisfied earlier (line 9).

Algorithm 4 SP(q, R, G, I, I^α)

1: MinHeap $H_k = \varnothing$, ordered by $f(L(T_p), S(q, p))$
2: **for each** keyword t_i in $q.\psi$ **do**
3:     Load posting list pl_i of t_i from I
4:     Load posting list of t_i from I^α
5: Construct $M_{q.\psi}$
6: $\theta = +\infty$
7: Queue $Q = (root)$
8: **while** e=GETNEXT(Q, R, q) **do**
9:     **if** $f_B^\alpha(e) \geq \theta$ **then break**
10:     **if** e refers to a place p **then**
11:         **if** e is Unqualified **then continue**   ▷ Pruning Rule 1
12:         $T_p = $ GETSEMANTICPLACEP$(q.\psi, p, G, M_{q.\psi})$
13:         **if** $L(T_p) == +\infty$ **then continue**
14:         Compute the ranking score f of T_p
15:         H_k.add(T_p, f)
16:         Update θ
17:     **else**   ▷ e refers to a node N
18:         **for** each entry e in N **do**
19:             Compute α-bound on the looseness $L_B^\alpha(T_e)$ for e
20:             Compute α-bound on the ranking score $f_B^\alpha(e)$ for e
21:             **if** $f_B^\alpha(e) < \theta$ **then**   ▷ Pruning Rules 3 and 4
22:                 Add $(e, f_B^\alpha(e))$ to Q
23: **return** H_k

---

6. EXPERIMENTS

To evaluate the performance of methods BSP (Section 3), SPP (Section 4), and SP (Section 5), we conducted an empirical study using real datasets, under various settings.

6.1 Settings

Datasets. We extracted the data used in our experiments from well-known real RDF knowledge bases, namely DBpedia and Yago (version 2.5). In DBpedia, there are 8,099,955 vertices and 72,193,833 edges in the directed RDF graph, with a dictionary of 2,927,026 unique words. The documents of all vertices are organized by an inverted index. The average posting list length is 56.46, which means on average, a word appears in the documents of 56.46 vertices in the graph. Among all vertices, 883,665 are places with coordinates. In Yago, there are 8,091,179 vertices and 50,415,307 edges in the directed RDF graph, with a dictionary of 3,778,457 distinct words. The documents of all vertices are organized by an inverted index with average posting list length 7.83. Among all the vertices, 4,774,796 vertices are places with coordinates. The original DBpedia and Yago data graphs are highly connected, with many edges representing "sameAs" "linksTo" and "redirectTo" relationships, which introduce semantically meaningless paths. In the datasets we use, such edges are removed. As a result, DBpedia consists of a huge weak connected component (WCC) with 8,099,624 vertices and 145 tiny WCCs with less than 10 vertices each. Similarly, the resulting Yago graph has a huge WCC with 8,091,094 vertices and 4 tiny WCCs with average size around 20.

Queries. Generating kSP query locations and keywords totally at random reduces the probability of obtaining any results. Therefore, we tried to generate meaningful kSP queries, by following the spatial and keyword distribution of the datasets. For each generated query, we randomly select a place p in the RDF graph and then randomly select the query location from a large range around this place. From p, we explore the RDF graph in BFS manner and randomly select at least $|q.\psi|/2$ and at most $|q.\psi| \times factor$ vertices that are reachable from p ($factor \geq 1$). If there are less than $|q.\psi|/2$ vertices reachable from p, p is discarded to avoid the case that the subgraph around the query location is too limited. In this case, we randomly select another place and repeat the whole process. Among the selected $[|q.\psi|/2, |q.\psi| \cdot factor]$ vertices, we randomly choose at most $|q.\psi|$ vertices, and $|q.\psi|$ keywords are extracted from the documents of these vertices as the query keywords. We set $factor = 2$ which gives flexibility with respect to $|q.\psi|$ and is large enough to obtain many connected vertices from p, but not too large to obtain faraway vertices, which are less semantically relevant to p.

Parameter settings. Performance is evaluated by varying the number of requested TQSPs k, the number of query keywords $|q.\psi|$, α of the α-radius based bounds, and also the data size for scalability evaluation. By default, $k = 5, |q.\psi| = 5, \alpha = 3$. We vary one parameter while fixing the other two. Specifically, we report the result when parameter k varies in $\{1, 3, 5, 8, 10, 15, 20\}$, $|q.\psi|$ varies in $\{1, 3, 5, 8, 10\}$, and α varies in $\{1, 3, 5\}$. For each setting, we run 100 queries and measure the average runtime, number of TQSP computations, and number of R-tree nodes accessed.

Platform. All methods were implemented in Java and evaluated on a 3.4 GHz quad-core machine running Ubuntu 12.04 with 16 GBytes memory. For the two datasets, the sizes of the R-trees, the RDF graphs, and the inverted indexes are shown in Table 4. The R-tree and the RDF graph are assumed to be memory-resident. Although the inverted indexes used can also fit the main memory, we choose to follow the setting of commercial search engines, where the inverted index is disk-resident. The reason is that for each query, only a small portion of the inverted index is relevant and needs be kept in main memory. Besides, such a design is scalable when more textual data added to the RDF knowledge base.

Preprocessing costs. The time costs of constructing the indexes and the data structures used for pruning are shown in Table 5. To construct the R-tree, we inserted the places in it one-by-one, in order to achieve better quality. The cost can be drastically reduced if bulk loading was used [45]. The inverted index for the documents of all the vertices in the RDF graph takes a few minutes only to construct. The TFlabel index [20] for reachability queries, which facilitates Pruning Rule 1, is also constructed within reasonable time. The DBpedia data are richer in terms of text, therefore the times to build the corresponding inverted and TFlabel indices are higher compared to those for Yago. For alpha-radius preprocessing, there are two dominant cost factors: (i) the computation of α-radius Word Neighborhood (WN) for each place and R-tree node, and (ii) building the inverted index of the α-radius WNs. For DBpedia, the average keyword frequency is 56.46, which renders the α-radius WNs of DBpedia to be larger than 32GB. To build the inverted index for so large data without exceeding the available memory, we had to create inverted indexes for parts of the data and merge them together in the end, which explains the high cost of constructing the inverted index for DBpedia.
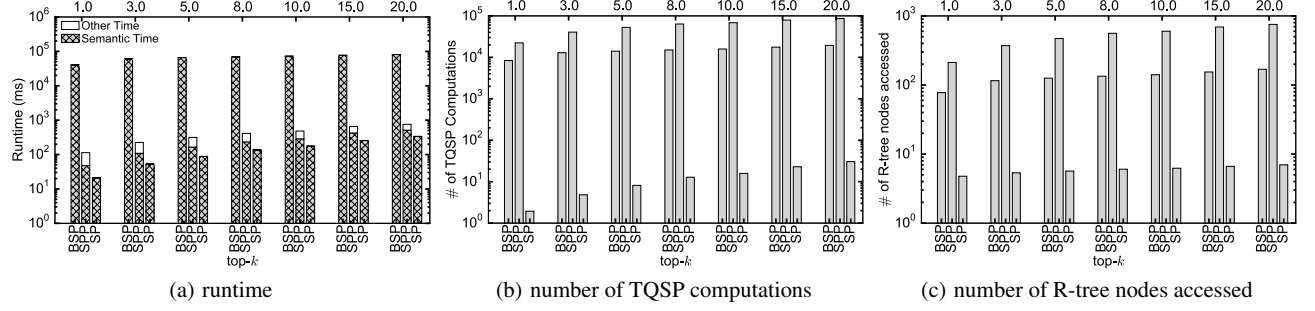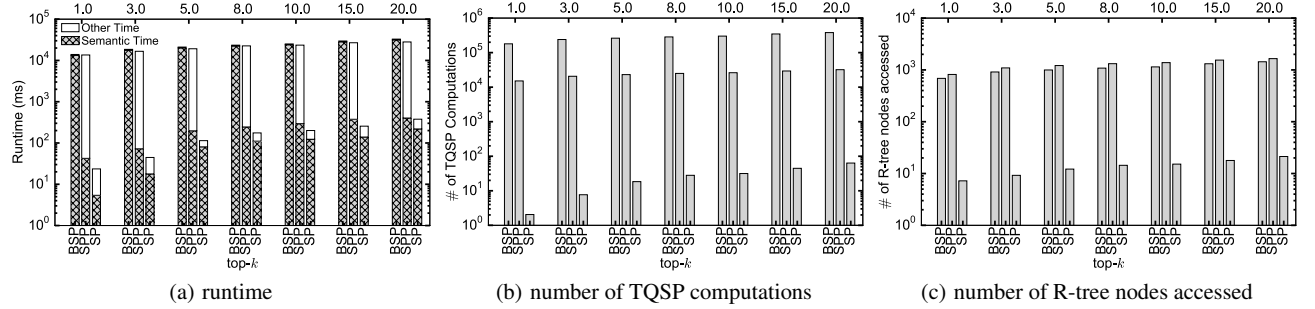
(a) runtime (b) number of TQSP computations (c) number of R-tree nodes accessed

Figure 3: Varying k on DBpedia.

(a) runtime (b) number of TQSP computations (c) number of R-tree nodes accessed

Figure 4: Varying k on Yago.

Table 4: Storage Cost

Data	R-tree	RDF graph	Inverted index
DBpedia	50.54MB	607.95MB	1307.98MB
Yago	273.17MB	454.81MB	231.91MB

Table 5: Preprocessing and indexing time (minutes)

Dataset	R-tree	Inverted index	TFlabel index	$\alpha(=3)$-radius
DBpedia	3.17	4.61	22.60	1192.01
Yago	31.90	1.00	6.09	101.61

6.2 Efficiency Evaluation

BSP takes too long to finish for some queries because (i) the termination condition (line 7 of Algorithm 1) only uses spatial distance as a (very loose) lower bound, (ii) function GETSEMANTIC-PLACE wastes computational cost for places that are not qualified semantic places, and (iii) function GETSEMANTICPLACE wastes time on the construction of the TQSPs that cannot be part of the kSP result. Hence, in our experiments, we set the maximum runtime for the queries using BSP to 120 seconds and abort those that take longer time.

6.2.1 Varying k.

Figures 3 and 4 show the cost of all methods on dataset DBpedia and Yago, respectively. As expected, the runtime, the number of TQSP computations, and the number of R-tree nodes accessed all increase as k increases, since a larger number of requested semantic places requires exploring a larger search space.

On dataset DBpedia (Figure 3), SP is 240-1865 times faster than BSP and 2-5 times faster than SPP for all k. The performance gap is maintained as k increases. The runtime of SP stays under 500ms for all values of k. For all the methods, the cost of constructing TQSPs dominates the runtime (shown as the "semantic

time" in Figure 3(a)). SPP includes other costs (i.e., "other time" in Figure 3(a)), which are mainly due to the reachability queries used in Pruning Rule 1. SP is the most efficient method in terms of both semantic time and other time, confirming the effectiveness of the proposed α-radius based bounds and Pruning Rules 3 and 4. SPP outperforms BSP because of Pruning Rules 1 and 2. As Figure 3(b) shows, SP only needs to compute the TQSPs for around 2-30 candidate places and accesses around 6 R-tree nodes on average, while SPP needs to compute tens of thousands TQSPs and access hundreds of R-tree nodes. Note that the numbers of TQSP computations and R-tree node accesses by BSP are smaller than the corresponding numbers by SPP, due to the 120 second time limit on BSP that that forces many queries to be terminated before finishing; this means that fewer places are processed in BSP compared to SPP, however, BSP may fail to return an answer, while SPP always computes the correct result. Furthermore, the runtime of SPP is much lower than that of BSP, which indicates that SPP takes less time to process more places than BSP does.

The results are similar on dataset Yago (Figure 4). Compared to DBpedia, the runtime gap between SPP and BSP decreases. However, the "semantic time" of SPP is 75-314 times less than the "semantic time" of BSP, which indicates that the pruning techniques in Section 4 reduce cost for TQSP computations significantly, but at the cost of performing reachability queries (i.e., the "other time" in Figure 4(a)). Yago contains more than 4.77M places, while DBpedia has 887K places. Therefore more reachability queries are issued on Yago compared to DBpedia, which leads to only a minor improvement of SPP over BSP on Yago. On the other hand, SP is robust in pruning a lot of places and nodes and achieves excellent performance on this large spatial RDF dataset.

6.2.2 Varying $|q.\psi|$.

Figure 5 compares the runtimes of all methods on DBpedia and Yago. In this and the subsequent experiments, we do not show the

number of TQSP computations and the number of R-tree nodes accessed by the methods for the interest of space and because they do not give different insights compared to the previous experiment. Generally, the runtimes of all methods increase with the number of query keywords $|q.\psi|$, since more vertices in RDF graph need to be explored to discover TQSPs covering all the query keywords in $|q.\psi|$. Again, SP is significantly faster than the other methods and the performance gap widens with $|q.\psi|$. Due to the larger number of places in Yago, which require more reachability queries processed in SPP, the runtime gap between SPP and BSP on Yago is smaller than that on DBpedia. However, recall that BSP is terminated after 2 minutes, so it fails to produce results for a number of queries, while SPP is always correct. SPP has much lower "semantic time" than BSP, however, it performs numerous reachability queries, which eventually dominate its runtime cost.
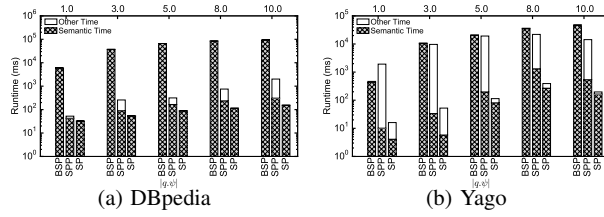

(a) DBpedia   (b) Yago

Figure 5: Varying $|q.\psi|$

6.2.3 Tuning α

In the next experiment, we evaluate the effect of parameter α in SP. Table 6 displays the total space that the α-radius word neighborhoods occupy, for the two datasets and different values of α. As expected, the space increases with α. On both datasets, the space is moderate when $\alpha = 1, 2, 3$, but increases rapidly to 204.70GB on DBpedia and 30.63GB on Yago when $\alpha = 5$.

α	1	2	3	5
DBpedia (GB)	3.56	24.33	32.53	204.70
Yago (GB)	1.07	3.61	12.37	30.63

Table 6: α-radius word neighborhood size

We evaluate the performance of SP with $k = 1, 3, 5, 8, 10, 15, 20$ when varying α from 1 to 5 on DBpedia and Yago (Figure 6). The number of query keywords is fixed to $|q.\psi| = 5$. Note that with a larger α, the exploring direction of SP is more biased to TQSP looseness than spatial distance (Lemmas 2 and 4). On DBpedia data, when changing α from 1 to 5, the runtime of SP decreases, since large α values enable tighter bound derivations and facilitate the pruning of more pruned places and nodes. We also observed that the number of TQSP computations and the number of R-tree nodes accessed significantly decrease when changing α from 1 to 3, but remain stable when changing α from 3 to 5.

Yago has keyword frequency 7.83, which is much smaller than that of DBpedia (56.46), meaning that it is generally more difficult to find a query keyword that can be reached from a place candidate to construct TQSPs. Recall that TQSP computation takes too much time and the exploration direction is biased to it; thus, a larger α may increase rather than decrease the runtime of kSP queries. This is confirmed by the findings shown in Figure 6(b). When changing α from 1 to 3, the runtime of SP decreases significantly; the larger α value enables tighter bound derivations and more pruned places. However, the runtime increases when changing α from 3 to 5. Overall, based on the evaluation of different α values on the two datasets, we conclude that $\alpha = 3$ is a good choice w.r.t., both

performance gains and the α-radius word neighborhood size (i.e., index size).
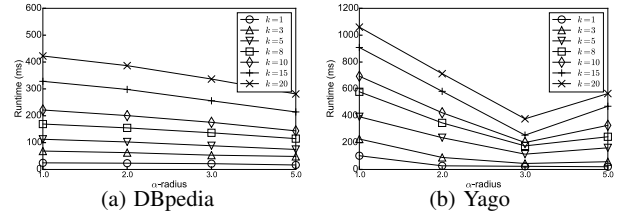

(a) DBpedia   (b) Yago

Figure 6: Varying α

6.2.4 Scalability

In this section, we evaluate the performance of the three methods on datasets of different sizes. We adopt the random jump sampling method [44] with probability $c = 0.15$ on the Yago dataset to generate RDF graphs of different sizes (described in Table 7). The associated documents of the selected vertices are also included in each generated dataset.

Figure 7 shows the performance of all methods as a function of the graph size. To be consistent, we generate queries using the smallest dataset and apply the generated queries on all datasets. The runtime of BSP and SPP generally increases but not dramatically with the graph size. On the other hand, the runtime of SP slightly decreases as the graph becomes larger. The reason behind this behavior, as we found out by analyzing the results, is that with more edges (larger graph), the connectivity is better, which can make it easier to find good TQSPs without exploring too many places. This is especially true for SP, which takes advantage of the pruning rules and the α-radius bounds to prune places that are not associated with the keywords early.

# of vertices	# of edges	# of places
2,000,000	11,659,509	1,144,705
4,000,000	24,174,226	2,317,671
6,000,000	36,966,773	3,507,942
8,091,179	50,415,307	4,774,796

Table 7: Datasets extracted from Yago by Random Jump
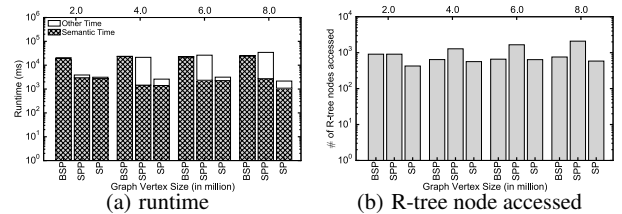

(a) runtime   (b) R-tree node accessed

Figure 7: Varying graph size by random jump sampling (Yago).

6.2.5 Results with Large Looseness

In order to evaluate the robustness of SP for (hard) queries having results of large looseness, we conducted a set of experiments, in which the generated queries have these characteristics. For this purpose, we generated two types of queries: *Small distance large looseness* (SDLL) kSP queries have as results places that are near the query location and have large looseness; *Large distance large looseness* (LDLL) kSP queries have as results places that are quite far from the query location and have large looseness. To generate SDLL and LDLL queries, we followed a similar methodology as

the one described in Section 6.1, with the following differences. First, in SDLL (LDLL) queries the query location is chosen to be near (far) from the place $p = (x, y)$ used to generate the query keywords. Specifically, in LDLL queries, the query location is a distant point location $(x, y + 90)$, increasing p's longitude by 90 degrees. Second, for both SDLL and LDLL queries, we choose infrequent words (with term frequency < 100) beyond 4 hops from p in the RDF graph as the query keywords. Due to (i) the low frequency of the selected words and (ii) the fact that similar places tend to be collocated [17, 18], the generated queries are anticipated to have as results places in the spatial neighborhood of p and with similar looseness as p (i.e., large looseness).

In order to confirm the correctness of our SDLL and LDLL query generators, we generated 100 queries in each of the two classes as well as 100 queries using the original query generator (Section 6.1). Then, for each query class (SDLL, LDLL, and O for our original generator), we averaged the spatial distance and the looseness of their top-k results, as shown in Figure 8. Observe, that the statistics are consistent with the intent of the generators. The results of SDLL and LDLL queries have smaller and larger, respectively, spatial distance compared to the results of O queries. At the same time, both SDLL and LDLL queries return results of much larger looseness compared to O queries.
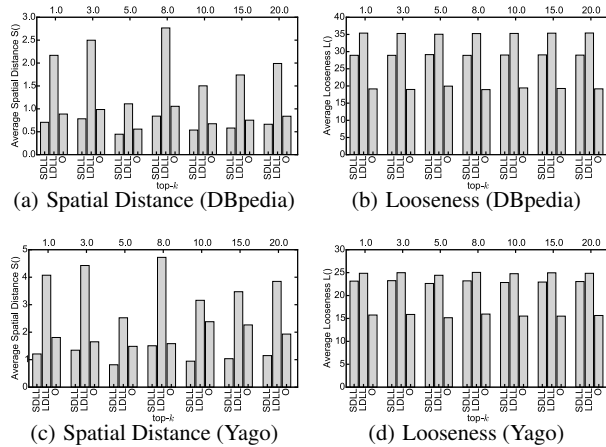

(a) Spatial Distance (DBpedia) (b) Looseness (DBpedia)

(c) Spatial Distance (Yago) (d) Looseness (Yago)

Figure 8: Average spatial distance and average looseness of the results by three classes of queries.

Figure 9 shows the runtime of BSP, SPP, SP for SDLL and LDLL kSP queries as a function of k. The relative performance of the algorithms is consistent with the previous experiments; SP is superior to SPP and outperforms BSP significantly. SDLL have similar cost as LDLL queries (slightly lower), which indicates that the dominant cost factor is not the spatial distance of the results, but their looseness. This can be also confirmed by comparing Figure 9 with Figure 3, which shows the evaluation cost of the original queries. For example, SP is 5-11 times slower on SDLL and LDLL queries compared to O queries. Still, SP achieves sub-second runtimes for k=5 even for these harder query classes and outperforms BSP by orders of magnitude.

6.2.6 Comparison with top-k aggregation

BSP and its optimized versions (SPP and SP) examine places in increasing spatial distance from the query location and compute their looseness as necessary, until the top-k places are confirmed. It is also possible to evaluate kSP queries by a hybrid approach that

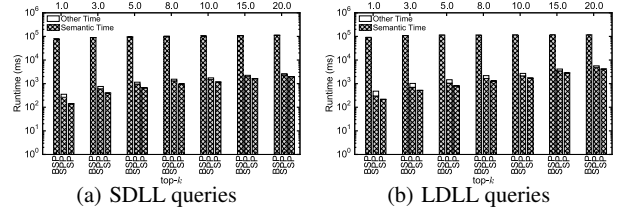

(a) SDLL queries (b) LDLL queries

Figure 9: Runtime of large-looseness queries (DBpedia).

combines two ranked lists of places: one that has qualified semantic places in increasing order of their looseness and one that has places in increasing order of their spatial distance to the query location. The first list can be incrementally generated by the extending the bottom-up RDF keyword search approach [43] (described in the Introduction) and the second by spatial nearest neighbor search. The two lists can be combined fast using the classic threshold algorithm (TA) of [25]: each time the next place is found by keyword search, its spatial distance is computed on-the-fly to complete its score; each time the next spatially nearest place is accessed, whether it is a qualifying semantic place (and its looseness) is computed by calling Algorithm 2. The algorithm terminates if the top-k TQSPs found so far cannot be outranked by the best possible place not found yet, according to the last incrementally computed spatial distance and looseness; i.e., the termination *threshold* of TA can obtained by applying f on these two values.

We implemented TA and compared it with our methods in Figure 10 for queries with various numbers of keywords $|q.\psi|$. On DBpedia, only when $|q.\psi| = 1$ TA performs better than BSP while being 8 times slower than SP. When $|q.\psi| \geq 3$, the runtime of TA increases significantly and TA becomes even slower than BSP. When $|q.\psi| \geq 3$, in order to find the semantic places in increasing looseness order, TA needs to start exploration from all the vertices containing any of the keywords and maintains $|q.\psi|$ queues to decide which vertex to explore next. TA also book-keeps for each vertex all the query keywords that have reached it (if a place has been reached by all query keywords, it becomes a semantic place and its looseness is calculated). These operations dominate the cost of TA, which spends a long time to rank the places by looseness. The results on Yago are similar to the DBpedia results. In addition, note that TA is slower than BSP for $|q.\psi| \geq 3$ for any value of k.

In summary, while it is extremely cheap to compute spatial distances and conduct spatial nearest neighbor search, it is expensive to conduct graph browsing and incremental ranking of places by looseness. This imbalance between the costs of computing spatial distance and looseness motivated the design of our algorithms, which prioritize the examination of places based on their spatial distances in order to minimize graph traversal operations.
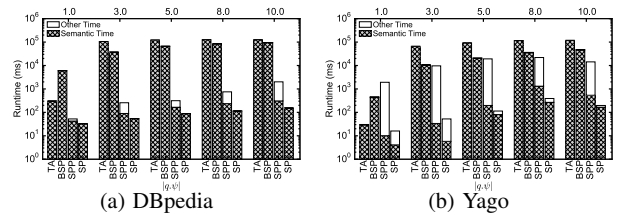

(a) DBpedia (b) Yago

Figure 10: Comparison with top-k aggregation (TA)

7. RELATED WORK

Keyword search on graph data. Due to its user-friendly query interface, keyword search is not only the de facto information retrieval method for the WWW data but also a popular querying mechanism for XML documents [21, 28], relational databases [13, 36], and graph data [22, 31, 41]. Traditional graph search algorithms convert queries into search over feature spaces, such as paths [51], frequent-patterns [57], and sequences [38], which focus more on the structure of the graph rather than the semantic content of the graph. Nevertheless, keyword search over graph data [13, 21, 22, 28, 31, 36, 41] determines a group of densely linked nodes in the graph by making use of both the content and the linkage structure. The overall quality of the results can be improved thanks to the re-enforcement between these two sources of information. Moreover, unexpected and interesting answers that are often difficult to be obtained via rigidly-formatted structured queries may be discovered by the keyword search. A recent survey about keyword search on schema graphs (e.g., relational data and XML documents) and schema-free graphs can be found in [56].

Keyword search on RDF data. RDF data are a special type of graph data, traditionally queried using structured query languages, like SPARQL. Recently, there has been increasing interest in keyword queries over RDF data. SPARQL queries are augmented with keywords for ranked retrieval of RDF data [24]. A keyword-based retrieval model over RDF graphs [23] identifies a set of maximal subgraphs whose vertices contain the query keywords. These subgraphs are ranked based on statistical language models (LMs) [49]. Top-k exploration of query candidates over RDF [52] first constructs a set of k query subgraphs based on the query keywords, and then let users choose the appropriate query graph. Query evaluation is performed using the underlying database engine. For the scalable and efficient processing of keyword queries on large RDF graphs, a summarization algorithm with pruning mechanisms on exploratory keyword search and its results is proposed [43]. Both [52] and [43] follow the definition of BLINKS [31] for the result subgraphs. k-nearest keyword (k-NK) search on RDF graphs [47] finds the k closest pairs of vertices, (v_i, u_i) that contain two given keywords q and w, respectively. SemDIS [30] demonstrates a system, where semantic associations are discovered in a large semantic metabase represented in RDF. Keyword query interpretation [26] personalizes the interpretation of a new query on RDF databases by a sequence of structured queries that correspond to the interpretations of keyword queries in the query history. Personalized keyword search on RDF [27] can personalize ranks based on the Ranking SVM approach that trains ranking functions with RDF-specific training features and utilizes historical user feedback. Diversified keyword search on RDF graphs [15] diversifies results by considering both the content and the structure of the results, as well as the RDF schema. A path-oriented RDF index for keyword search query [19] captures associations across RDF paths for improving the query execution performance.

Spatial RDF stores. Recently, there are many efforts toward the efficient storage and indexing of spatial RDF data. Parliament [8] is an implementation of GeoSPARQL. Strabon [42] employs a column-store approach to manage the RDF data in PostGIS, implementing two SO and OS indices for each property table, and uses spatial indexes on top of them. Brodt et al. [16] adopts a two-stage algorithm that either processes the non-spatial query components first and then verifies the spatial ones or the other way around to support spatial querying on RDF data. Geo-Store [54] uses a Hilbert space-filling curve to index the space and supports spatial range queries and NN search. S-Store [55] primarily indexes spatial RDF data based on their structure and uses their spatial locations to prune triples during search. In addition, several commercial systems, such as Oracle, Virtuoso [11], and OWLIM-SE [7], support spatial RDF data management, however, details about their internal design are not available. Recently, a spatial encoding scheme for RDF stores that supports efficient spatial data management was proposed in Liagouris et al. [46]. To our knowledge, no previous work on spatial RDF data management supports queries that combine spatial and keyword search.

Discussion. Our work differs significantly from existing work. Firstly, the kSP queries studied are schema-free; thus, query processing and optimization techniques in traditional RDF stores are inapplicable. Secondly, although keyword search on RDF data (i.e., one aspect of a kSP query) has been investigated in the literature, there is no direct way of extending the existing algorithms to process kSP queries. In fact, extensions are expected to be highly inefficient because they do not guide search based on the query location, as discussed in the beginning of Section 3. Thirdly, kSP queries enable users to search nearby places that semantically match their preferences (expressed by keywords). Such a functionality finds important and useful applications as discussed in Section 1, and cannot be achieved by other forms of queries in the literature. Fourthly, the query evaluation algorithms proposed in this paper are orthogonal to indexing and storage techniques for graph data, which can be applied to further improve the performance of kSP search on very large RDF data.

8. CONCLUSION

In this paper, we proposed a top-k relevant semantic place retrieval (kSP) query that takes as input a query location $q.\lambda$ and a set of query keywords $q.\psi$, and returns the top-k tightest qualified semantic places ranked by their spatial distance to $q.\lambda$ and their semantic looseness to $q.\psi$ over the RDF graph. kSP queries do not require the use of any structured query languages, thus they are user-friendly and do not rely on the knowledge of the RDF schema. Compared to existing keyword search, kSP queries are spatial-aware and support spatial-personalized search. After suggesting a baseline algorithm (BSP), we propose two pruning techniques that reduce the cost of computing the semantic relevance of each place to the query keywords; namely (i) an *unqualified place pruning* approach that discards places which do not cover the query keywords without computing their TQSPs, and (ii) a *dynamic bound based pruning* approach that early terminates the TQSP computation of a place if the place cannot enter the kSP result. To further boost efficiency, in Section 5, we introduce the concept of α-radius word neighborhood and propose α-radius bounds on both the looseness and the ranking scores that can be applied to prune not only individual places but also sets of places (i.e. R-tree nodes). The proposed techniques are evaluated on two large real RDF data sets, i.e., DBpedia and Yago. The results show that applying all techniques enables processing kSP in less than a second for most settings and outperforms the basic method by orders of magnitude.

In this paper, the RDF graph data assumed to be memory-resident. In the future, we plan to integrate and extend existing indexing and storage techniques for disk-resident graph data to develop a scalable solution. Our kSP definition follows directly from previous work of RDF keyword search [26, 31, 43, 52], where only incoming paths from keywords to the root node of a result subgraph are considered. In the future, we also plan to qualitatively evaluate an alternative definition of semantic places, where the keywords in outgoing paths from them are also considered (i.e., edge directions are disregarded).

9. REFERENCES

[1] Alternative fueling station locator. http://www.afdc.energy.gov/locator/stations/.

[2] Bbc lab post. http://www.bbc.co.uk/blogs/internet/entries/ 63841314-c3c6-33d2-a7b8-f58ca040a65b.

[3] Crime in chicagoland. http://crime.chicagotribune.com/.

[4] Data.gov. http://www.data.gov.

[5] Dbpedia. http://wiki.dbpedia.org.

[6] Hospital compare. http://health.data.gov/def/cqld.

[7] Owlim-se. http://owlim.ontotext.com/display/OWLIMv43/OWLIM-SE.

[8] Parliament. http://parliament.semwebcentral.org.

[9] Patients like me. www.patientslikeme.com.

[10] Spot crime. http://www.spotcrime.com/.

[11] Virtuoso. http://virtuoso.openlinksw.com.

[12] Yago. http://www.mpi-inf.mpg.de/departments/ databases-and-information-systems/research/yago-naga/ yago/.

[13] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[14] R. Battle and D. Kolas. Enabling the geospatial semantic web with parliament and geosparql. *Semantic Web*, 3(4):355–370, 2012.

[15] N. Bikakis, G. Giannopoulos, J. Liagouris, D. Skoutas, T. Dalamagas, and T. Sellis. Rdivf: Diversifying keyword search on RDF graphs. In *TPDL*, pages 413–416, 2013.

[16] A. Brodt, D. Nicklas, and B. Mitschang. Deep integration of spatial query processing into native RDF triple stores. In *SIGSPATIAL*, pages 33–42, 2010.

[17] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3(1):373–384, 2010.

[18] X. Cao, G. Cong, C. S. Jensen, and M. L. Yiu. Retrieving regions of interest for user exploration. *PVLDB*, 7(9):733–744, 2014.

[19] P. Cappellari, R. D. Virgilio, A. Maccioni, and M. Roantree. A path-oriented RDF index for keyword search query processing. In *DEXA*, pages 366–380, 2011.

[20] J. Cheng, S. Huang, H. Wu, and A. W. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD*, pages 193–204, 2013.

[21] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. Xsearch: A semantic search engine for XML. In *VLDB*, pages 45–56, 2003.

[22] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *PVLDB*, 1(1):1189–1204, 2008.

[23] S. Elbassuoni and R. Blanco. Keyword search over RDF graphs. In *CIKM*, pages 237–242, 2011.

[24] S. Elbassuoni, M. Ramanath, R. Schenkel, and G. Weikum. Searching RDF graphs with SPARQL and keywords. *IEEE Data Eng. Bull.*, 33(1):16–24, 2010.

[25] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

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

[27] G. Giannopoulos, E. Biliri, and T. Sellis. Personalizing keyword search on RDF data. In *TPDL*, pages 272–278, 2013.

[28] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: ranked keyword search over XML documents. In *SIGMOD*, pages 16–27, 2003.

[29] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[30] C. Halaschek-Wiener, B. Aleman-Meza, I. B. Arpinar, and A. P. Sheth. Discovering and ranking semantic associations over a large RDF metabase. In *VLDB*, pages 1317–1320, 2004.

[31] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[32] J. A. Hendler, J. Holm, C. Musialek, and G. Thomas. US government linked open data: Semantic.data.gov. *IEEE Intelligent Systems*, 27(3):25–31, 2012.

[33] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[34] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, 194:28–61, 2013.

[35] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[36] V. Hristidis and Y. Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[37] J. Inglis. Inverted indexes and multi-list structures. *Comput. J.*, 17(1):59–63, 1974.

[38] H. Jiang, H. Wang, P. S. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *ICDE*, pages 566–575, 2007.

[39] R. Jin, N. Ruan, S. Dey, and J. X. Yu. SCARAB: scaling reachability computation on large graphs. In *SIGMOD*, pages 169–180, 2012.

[40] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Database Syst.*, 36(1):7, 2011.

[41] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[42] K. Kyzirakos, M. Karpathiotakis, and M. Koubarakis. Strabon: A semantic geospatial DBMS. In *ISWC*, pages 295–311, 2012.

[43] W. Le, F. Li, A. Kementsietsidis, and S. Duan. Scalable keyword search on large RDF data. *TKDE*, 26(11):2774–2788, 2014.

[44] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *KDD*, pages 631–636, 2006.

[45] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for R-tree packing. In *ICDE97*, pages 497–506, 1997.

[46] J. Liagouris, N. Mamoulis, P. Bouros, and M. Terrovitis. An effective encoding scheme for spatial RDF data. *PVLDB*, 7(12):1271–1282, 2014.

[47] X. Lian, E. D. Hoyos, A. Chebotko, B. Fu, and C. Reilly. k-nearest keyword search in RDF graphs. *J. Web Sem.*, 22:40–56, 2013.

[48] T. Neumann and G. Weikum. RDF-3X: a risc-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[49] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *SIGIR*, pages 275–281, 1998.

[50] E. Prud'Hommeaux, A. Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

[51] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.

[52] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *ICDE*, pages 405–416, 2009.

[53] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD*, pages 913–924, 2011.

[54] C. Wang, W. Ku, and H. Chen. Geo-store: a spatially-augmented SPARQL query evaluation system. In *SIGSPATIAL*, pages 562–565, 2012.

[55] D. Wang, L. Zou, Y. Feng, X. Shen, J. Tian, and D. Zhao. S-store: An engine for large RDF graph integrating spatial information. In *DASFAA*, pages 31–47, 2013.

[56] H. Wang and C. C. Aggarwal. A survey of algorithms for keyword search on graph data. In *Managing and Mining Graph Data*, pages 249–273. 2010.

[57] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.

[58] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: scalable reachability index for large graphs. *PVLDB*, 3(1):276–284, 2010.

[59] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4):265–276, 2013.

[60] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: Answering SPARQL queries via subgraph matching. *PVLDB*, 4(8):482–493, 2011.