The VLDB Journal (2020) 29:893–917
https://doi.org/10.1007/s00778-019-00591-8

REGULAR PAPER

Top-*k* relevant semantic place retrieval on spatiotemporal RDF data

Dingming Wu[1] · **Hao Zhou**[1] · **Jieming Shi**[2] · **Nikos Mamoulis**[3]

Received: 27 November 2018 / Revised: 11 September 2019 / Accepted: 2 November 2019 / Published online: 19 November 2019

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 users from these requirements; users only input a set of keywords, and the goal is to find small RDF subgraphs that contain all keywords. At the same time, popular RDF knowledge bases also include spatial and temporal semantics, which opens the road to spatiotemporal-based search operations. In this work, we propose and study novel keyword-based search queries with spatial semantics on RDF data, namely *k*SP queries. The objective of the *k*SP query is to find RDF subgraphs which contain the query keywords and are rooted at spatial entities close to the query location. To add temporal semantics to the *k*SP query, we propose the *k*SPT query that uses two ways to incorporate temporal information. One way is considering the temporal differences between the keyword-matched vertices and the query timestamp. The other way is using a temporal range to filter keyword-matched vertices. The novelty of *k*SP and *k*SPT queries is that they are spatiotemporal-aware and that they do not rely on the use of structured query languages. We design an efficient approach containing two pruning techniques and a data preprocessing technique for the processing of *k*SP queries. The proposed approach is extended and improved with four optimizations to evaluate *k*SPT queries. Extensive empirical studies on two real datasets demonstrate the superior and robust performance of our proposals compared to baseline methods.

Keywords Semantic place · RDF data · Spatiotemporal data

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 [4] and YAGO [11] 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 and objects) which are linked to other entities, types, literals, or descriptions. 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 attributed graph, where nodes are entities, attributes are types/literals, and the edges are predicates which describe the relationships between nodes.

The English version of DBpedia currently describes 6.0M entities, roughly including 1.5M persons, 810K places, 490K creative works, 275K organizations, 301K species, etc.

This work was supported in part by Grant No. 2019A1515011721 from Natural Science Foundation of Guangdong, China and by Grant No. 61502310 from National Natural Science Foundation of China and by Grant No. 17253616 from Hong Kong RGC.

✉ Jieming Shi
shijm@nus.edu.sg

Dingming Wu
dingming@szu.edu.cn

Hao Zhou
zhouhao2017@email.szu.edu.cn

Nikos Mamoulis
nikos@cs.uoi.gr

[1] College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China

[2] School of Computing, National University of Singapore, Singapore, Singapore

[3] Department of Computer Science and Engineering, University of Ioannina, Ioannina, Greece

YAGO includes more than 10M entities (like persons, organizations, cities) and contains more than 120M facts about these entities. Data.gov [3] is the largest open-government, data-sharing Web site 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 [30], e.g., Hospital Compare [5], Patients Like Me [8], Alternative Fueling Station Locator [1], Crime in Chicagoland [2], SpotCrime [9].

Keyword search on RDF data RDF data are traditionally accessed with the help of a structured query language, like SPARQL [46,53,62]. 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 [20,42,56]. This model allows users to retrieve information from RDF knowledge bases without the direct use of SPARQL-like languages and without the 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 edges. For the purpose of keyword search, this graph can be simplified [42] 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 or some 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 defines a *looseness* score for the subgraph [23,42,56]. 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 [33] and general keyword search on graphs [29].

Spatiotemporal RDF data RDF data have been enriched to include additional semantics. For example, YAGO2 [32] 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 spatiotemporal-based retrieval. To fully utilize spatially enriched RDF data, the GeoSPARQL standard [13], defined by the Open Geospatial Consortium (OGC), extends RDF and SPARQL to represent geographic information and support spatial queries. RDF stores such as Virtuoso [10], Parliament [7], Strabon [41] are developed to support GeoSPARQL features. Liagouris et al. [44] extended the RDF-3X data store [49] such that the locations of spatial entities are encoded into their IDs; this facilitates efficient evaluation of spatial search operations in GeoSPARQL queries. To support spatiotemporal-oriented applications on RDF data, stRDF [40] extends RDF with the ability to represent spatial and temporal data and stSPARQL [40] extends SPARQL for querying stRDF data. The g^{st}-store [60] system has been designed for large RDF graphs integrating spatial and temporal information. 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 and *k*SPT queries** In this paper, we propose two novel ways of searching spatial temporal RDF data, namely the *top-k relevant semantic place retrieval* (*k*SP) query, which combines keyword search with location-based retrieval and the *top-k relevant semantic place with temporal constraint retrieval* (*k*SPT) query, which adds a temporal constraint to the *k*SP query. Both *k*SP and *k*SPT 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. On the other hand, the *k*SP and the *k*SPT query have the following unique features compared to RDF keyword search. These two types of queries retrieve *semantic places*, i.e., only subgraphs rooted at a place entity and are query-location-aware. In addition, the *k*SPT query is query-temporal-aware.

Applications Both the *k*SP and the *k*SPT query find a number of interesting applications. For instance, the *k*SP query 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. The aim of a *k*SPT query is to retrieve nearby places that are not only relevant to the search keywords but also satisfy the given temporal constraint. It can be used by journalists who want to find nearby places where recent information or the information for a certain time period could be found, marketing managers who need to investigate nearby places to collect the recent sales information of one type of product, and a data analyst who may want to look for historical information related to some time period in a knowledge base.

A *k*SP query takes a query location, a set of query keywords, and the number k of requested places as arguments, and returns as result the top-k *tightest qualified semantic places* (TQSP) according to a scoring 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 with a spatial location, e.g., via hasGeometry predicates) and (ii) the documents associated with all the vertices in the tree collectively cover all query keywords. In accordance with existing work on RDF keyword search [20,42,56], 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).

The kSPT query extends the kSP query by considering temporal semantics of the entities in RDF graph. We introduce two variants of the kSPT query, i.e., the kSPT^d query and the kSPT^r query. Besides the three arguments in the kSP query, the kSPT^d query has a timestamp and the kSPT^r query has a temporal range. The kSPT^d query returns the top-k TQSPs according to a scoring function that considers both the spatial distance to the query location and the temporal difference-based looseness (TDL). The kSPT^r query retrieves k TQSPs with the smallest combined temporal range-based looseness (TRL) and spatial distance to the query location. Both TDL and TRL are variants of the looseness of a qualified semantic place (QSP). TDL assigns weights (e.g., difference between the query timestamp and the timestamp of keyword-matched nodes) to the graph distances between the place (root) and the occurrences of the covered keywords at the nodes. TRL considers the nodes whose timestamps belong to the given temporal range when constructing semantic places.

Data representation and indexing Our previously published work [55] is the first work that proposes and studies kSP queries, and this paper extends it and proposes kSPT queries; therefore, no other existing system and algorithmic support for kSP and kSPT query evaluation. Typical RDF stores are designed for SPARQL queries; however, kSP and kSPT queries require graph browsing and search operations (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 [65]. 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 fast finding of the vertices that contain given keywords in their documents. Second, all place vertices are spatially indexed by an R-tree [26], 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 [29,42]. 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 [29,42].

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., [29,42]) is inefficient. As an alternative, we propose a basic semantic place 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 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 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 semantic place retrieval with pruning (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 word neighborhoods (in the RDF

[1] Disk-based graph representations for RDF data (e.g., [67]) can also be used for larger-scale data.

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 Semantic Place 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.

To evaluate kSPT queries, the proposed algorithm SP is extended by incorporating the temporal constraint checking and calculation into the TQSP construction. Specifically, dynamic bounds on temporal constraint-based looseness (Sect. 4.1.2) are derived to prune places by aborting their TQSP computation early. The α-radius-based bounds (Sect. 4.1.3) are introduced to help pruning place vertices and the R-tree nodes. However, as shown in our experiment, on DBpedia data, it takes hundreds of milliseconds to evaluate a kSP query using algorithm SP, while it takes tens of seconds to compute a kSPT query using the extended version of algorithm SP. The performance gap is orders of magnitude. *The challenges lie in three aspects* First, dynamic bounds and the α-radius-based bounds are derived based on the minimum temporal difference between the query timestamp and the timestamps of the vertices whose documents contain term w, denoted as $d_m^t(q.\delta, w)$. However, the bounds are loose when the vertex whose timestamp is used to compute $d_m^t(q.\delta, w)$ is not reachable from a place or a node, which means that pruning is not effective. Second, the reachability test is used multiple times in the algorithm and its performance differs from word to word. Materializing reachable vertices for expensive words is expected to save the computation cost. However, the external storage needed grows rapidly, and it will incur I/O cost that is also time inefficient. Third, loading the α-radius temporal word neighborhood is time-consuming, since all the lengths of the shortest paths from places/nodes to words within the α-radius are stored, together with the corresponding timestamps. Having observed the above disadvantages of the extended algorithm SP for the kSPT query, four optimizations are proposed: (i) tighter bounds on temporal constraint-based looseness are derived, (ii) materializing reachable vertices for frequent words to reduce the cost incurred by reachability tests, (iii) downsizing the α-radius temporal word neighborhoods so that the cost of loading long posting lists from disk is reduced, and (vi) a pruning rule for R-tree nodes which contains multiple places is introduced.

A preliminary version of this paper has appeared in [55]. In that work, we only defined and studied kSP queries. Here, besides introducing temporal constraints and extending algorithm SP to evaluate kSPT queries, we introduce four optimizations that help further improve the performance of the query evaluation.

Outline Section 2 introduces the definition of the kSP and the kSPT queries and relevant concepts. Algorithms for evaluating kSP queries are presented in Sect. 3 and algorithms for processing kSPT queries are introduced in Sect. 4. Our empirical study is reported in Sect. 5. Related work is reviewed in Sect. 6, and we conclude in Sect. 7.

2 Problem definition

A spatiotemporal 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 with 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. Some of the entities are associated with timestamps. Each RDF triple corresponds to a directed edge from an entity (subject) to another entity (object). In accordance with 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 subtree 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 with multiple semantics by being connected to different vertices.

Example 1 Figure 1 shows the graph representation of several triples extracted from DBpedia. The squares are place vertices and the circles are non-place vertices in the RDF graph, representing entities. The edges (labeled by predicates) model the relationships between entities. Each entity is associated with a textual description (document) extracted from its URI, predicates, and literals [42]. Table 1 displays the documents of all vertices in Fig. 1 (due to space constraints, for each document, only some of the terms are shown). The spatial coordinates of p_1 and p_2 in Fig. 1 are shown in Fig. 2. Vertices v_2 and v_3 have timestamps. The tree consisting of vertices $\{p_1, v_1, v_2\}$ rooted at p_1 is a semantic place. The

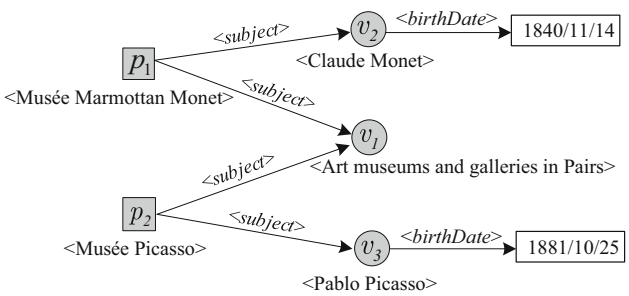


Fig. 1 RDF graph

Table 1 RDF Documents

p_1: {musée, marmottan, Monet}
v_1: {art, museum, galleries, Paris}
v_2: {Claude Monet, French, impressionist, painting}
v_3: {Pablo Picasso, Spanish, impressionist, painting}
p_2: {musée, Picasso}

Fig. 2 Map of places in Fig. 1 and query points

tree rooted at p_2 with vertices $\{p_2, v_1, v_3\}$ is another semantic place.

2.1 *k*SP query

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 k of requested semantic places. 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 (QSP)* Given a kSP query q and an 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 $\langle p, (v_1, v_2, \ldots) \rangle$, where p is the root and $(v_1, v_2, \ldots)$ includes all the other vertices. Given a kSP query, there may exist multiple semantic places with the same root p but different $(v_1, v_2, \ldots)$ sets. Following existing work on keyword search over graphs [29,42], we define the looseness of a qualified semantic place in Definition 2.

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

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 scoring 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 2 Assume that the given query keywords are $q.\psi = \{impressionist, art\}$. Based on the RDF example shown in Fig. 1, two qualified semantic places can be found, i.e., $\langle p_1, (v_1, v_2) \rangle$ and $\langle p_2, (v_1, v_3) \rangle$. The looseness of $\langle p_1, (v_1, v_2) \rangle$ is calculated by $1 + 1 + 1 = 3$, where $d_g(p_1, impressionist) = 1$, $d_g(p_1, art) = 1$.

Definition 3 *Top-k Relevant Semantic Place Retrieval* Given a kSP query q on a spatiotemporal RDF graph, the result of q includes k TQSPs minimizing scoring 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, and therefore, it appears 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. The proposed algorithms are directly applicable when the spatial network distance is used instead. In our algorithms, the places are retrieved in ascending order of Euclidean distance, using the R-tree. This module can be replaced by any existing algorithm that incrementally retrieves places based on the network distance [50].

Scoring function $f(L(T_p), S(q, p))$ can be any monotonic aggregate function which considers both $L(T_p)$ and $S(q, p)$. The kSP evaluation approaches proposed in this paper are independent to how f is defined. Without loss of generality, in the rest of the paper we use Eq. (1) as the scoring function. For normalization purposes, L_τ and S_τ are the maximum allowed looseness (e.g., $L_\tau = 50$) and the maximum allowed Euclidean distance (e.g., $S_\tau = 1000$ km), which means the Euclidean distance more than 1000 km is

[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.

the same as 1000 km and the semantic places with loosenesses larger than 50 are as loose as those whose loosenesses are 50.

$$f(L(T_p), S(q, p)) = \hat{L}(T_p) \times \hat{S}(q, p),$$
$$\hat{L}(T_p) = \frac{\min(L(T_p), L_\tau)}{L_\tau}, \quad \hat{S}(q, p) = \frac{\min(S(q, p), S_\tau)}{S_\tau}. \tag{1}$$

Example 3 Consider an example kSP query q with query location $q.\lambda = q_1$ as shown in Fig. 2 and query keywords $q.\psi = \{impressionist, art\}$. Places p_1 and p_2 are located at (48.86, 2.27) and (48.86, 2.36), respectively in Fig. 2. Based on the RDF graph in Fig. 1 and the documents in Table 1, given that $L_\tau = 10$ and $S_\tau = 10$; semantic place $T_{p_1} = \langle p_1, (v_1, v_2)\rangle$ has $S(q_1, p_1) = 0.014$ and $L(T_{p_1}) = 3$, $f(L(T_{p_1}), S(q_1, p_1)) = L(T_{p_1})/10 \times S(q_1, p_1)/10 = 0.00042$; semantic place $T_{p_2} = \langle p_2, (v_1, v_3)\rangle$ has $S(q_1, p_2) = 0.08$ and $L(T_{p_2}) = 3$, $f(L(T_{p_2}), S(q_1, p_2)) = 0.0024$. Therefore, T_{p_1} is returned as top-1 and T_{p_2} ranks second for the kSP query q.

If the query location of the kSP query q is changed to $q.\lambda = q_2$ and the query keywords are unchanged, T_{p_2} is returned as the top-1 semantic place and T_{p_1} ranks second.

2.2 *k*SPT query

The *top-k spatiotemporal semantic place retrieval* (kSPT) query extends the kSP query by considering the temporal dimension. A kSPT query q consists of four arguments: a query location $q.\lambda$, a set of query keywords $q.\psi$, a temporal argument, and the number k of requested semantic places. Besides the spatial proximity and the textual relevance considered in the kSP query, the kSPT query also takes the temporal proximity into account when ranking the candidate semantic places.

In particular, in the kSPT query, users can specify a subset of the query keywords in $q.\psi$ on which they wish to apply the temporal argument. The scoring function for the QSPs of the kSPT query then becomes $\gamma \cdot f(L^*(T_p), S(q, p)) + (1-\gamma) \cdot f(L(T_p), S(q, p))$ $(0 \le \gamma \le 1)$, where $L^*(T_p)$ is the looseness of the QSP computed by using only the query keywords for which we want to consider temporal information, and $L(T_p)$ is the looseness of the QSP using only the query keywords without temporal information, equivalent to the looseness in the kSP query. In the next section, we will explain how to compute $L^*(T_p)$ for different definitions of temporal relevance. In the rest of the paper, to make the presentation concise, we will assume that $\gamma = 1$. When $\gamma = 0$, the kSPT query reduces to the kSP query. The proposed algorithms can easily be adapted for the case where $0 < \gamma < 1$.

We now propose two variants of the kSPT query, i.e., the kSPT^d query that considers the temporal difference between the keywords in the QSPs and temporal argument of the query when constructing the QSPs and the kSPT^r query that retrieves the QSPs which satisfy a given temporal range. The specializations of $L^*(T_p)$ for these two variants are $L^d(T_p)$ and $L^r(T_p)$.

2.2.1 *k*SPT^d query

The temporal argument in the kSPT^d query q is a timestamp $q.\delta$. Next, we define the temporal difference-based looseness of a QSP in Definition 4.

Definition 4 *Temporal Difference-Based Looseness (TDL).* Given a QSP $T_p = \langle V', E'\rangle$, the minimum temporally weighted length of the path from root p to keyword $w_i \in q.\psi$ is

$$d_g^d(p, w_i) = \min_{v \in V' \wedge w_i \in v.\psi} \hat{d}(p, v) \times \hat{d}^t(q.\delta, v.\delta),$$
$$\hat{d}(p, v) = \frac{\min(1 + d(p, v), L_\tau)}{L_\tau},$$
$$\hat{d}^t(q.\delta, v.\delta) = \frac{\min(1 + d^t(q.\delta, v.\delta), D_\tau^t)}{D_\tau^t}, \tag{2}$$

where $d(p, v)$ is the length of the shortest path from p to v and $d^t(q.\delta, v.\delta)$ is the temporal difference[3] between the query timestamp $q.\delta$ and the timestamp of the vertex $v.\delta$. If vertex v has no timestamp, i.e., $v.\delta = \emptyset$, $d^t(q.\delta, v.\delta) = D_\tau^t$. D_τ^t and L_τ are used for normalization purposes. L_τ is the maximum allowed looseness that is the same as in Eq. 1. D_τ^t is the maximum allowed temporal difference (e.g., $D_\tau^t = 100$ days), which means the temporal difference more than 100 days is the same as 100 days. The temporal difference-based looseness of T_p is defined as $L^d(T_p) = \sum_{w_i \in q.\psi} d_g^d(p, w_i)/|q.\psi|$.

TDL aggregates the proximity of the query keywords in a QSP in terms of both the graph distance and the temporal difference. TDL favors the nearby vertices on the graph that not only match the query keywords but also are temporally close to the query timestamp. For normalization purposes, we add 1 to both $d(p, v)$ and $d^t(q.\delta, v.\delta)$ (as we will see later, the case of $L^d(T_p) = 0$ should be avoided in our scoring function for kSPT^d results). The smaller the TDL, 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 TQSP for the given query keywords, which is the QSP rooted at p with the smallest TDL.

Example 4 Consider a kSPT^d query with keyword $q.\psi = \{Spanish, impressionist\}$, timestamp $q.\delta =$ "1881/10/20",

[3] The temporal difference could be measured in days, minutes, etc.

and query location q_2 in Fig. 2. Based on the spatiotemporal RDF example shown in Fig. 1, given $L_\tau = 10$ and $D^t_\tau = 10$, a QSP is $T_{p_2} = \langle p_2, (v_3)\rangle$. The TDL of T_{p_2} is $(1+1+1)/10 \times (1+5)/10 = 0.9$, where $d(p_2, v_3) = 1$, $d^t(q.\delta, v_3.\delta) = 5$ days.

Definition 5 *Top-k Spatiotemporal Semantic Place Retrieval Based on Temporal Difference* Given a kSPT^d query q on a spatiotemporal RDF graph, the result of q includes k TQSPs minimizing scoring function $f(L^d(T_p),\ S(q,\ p))$, where $S(q,\ p)$ is the spatial distance between the query location and the root of the semantic place. Scoring function $f(L^d(T_p),\ S(q,\ p))$ is defined as

$$f(L^d(T_p), S(q,\ p)) = L^d(T_p) \times \hat{S}(q,\ p), \tag{3}$$

where $\hat{S}(q,\ p)$ is the normalized Euclidean distance defined in Eq. 1.

Recall that each place p has a unique TQSP, and therefore, it appears at most once in a kSPT^d result. The kSPT^d query aims at finding the TQSPs 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 based on both the graph distance and the temporal difference.

2.2.2 kSPT^r query

The temporal argument in the kSPT^r query q is a temporal range $q.r$. Based on $q.r$, the keywords in $q.\psi$ are divided into two groups: $q.\psi^r$ (temporal range relevant) and $q.\psi^{\bar{r}}$ (temporal range irrelevant), such that $q.\psi^r \cap q.\psi^{\bar{r}} = \emptyset$ and $q.\psi^r \cup q.\psi^{\bar{r}} = q.\psi$. A keyword w_i in $q.\psi$ is assigned to $q.\psi^r$, if there exist some vertices whose documents contain w_i and whose timestamps belong to the query temporal range $q.r$. A keyword w_j in $q.\psi$ is assigned to $q.\psi^{\bar{r}}$, if all the vertices whose documents contain w_i either do not have timestamps or have timestamps *only* outside the query temporal range $q.r$. Next, we define the temporal range-based looseness of a QSP in Definition 6.

Definition 6 *Temporal Range-Based Looseness (TRL)* Given a QSP $T_p = \langle V', E'\rangle$, let $L(T_p, W)$ be the looseness of T_p for keyword set W, i.e., $L(T_p, W) = \sum_{w_i \in W} d_g(p, w_i)$. The temporal range-based looseness of T_p is defined as

$$L^r(T_p) = 1 + \beta \cdot \frac{\min(L(T_p, q.\psi^r), L_\tau)}{L_\tau} + (1-\beta) \cdot \frac{\min(L(T_p, q.\psi^{\bar{r}}), L_\tau)}{L_\tau},$$

where β is used to adjust the importance of the temporal range relevant keywords.

The TRL aggregates the proximity of the query keywords and favors the temporal range relevant keywords. Parameter β is recommended to be set to 0.2, which means that the temporal range relevant keywords are given higher weight. It also can be modified when necessary in real applications. Constant 1 and L_τ in $L^r(T_p)$ are for normalization purposes. The smaller the TRL, 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 TQSP for the given query keywords, which is the QSP rooted at p with the smallest TRL.

Example 5 Consider a kSPT^r query with keyword $q.\psi = \{Spanish, art\}$, temporal query range $q.r$ = "[1881/01/01, 1881/12/30]", and query location q_2 in Fig. 2, given $L_\tau = 10$ and $\beta = 0.2$. Based on the spatiotemporal RDF example shown in Fig. 1, a QSP is $T_{p_2} = \langle p_2, (v_1, v_3)\rangle$. The TRL of T_{p_2} is calculated by $1+0.2\times 1/10+0.8\times 1/10 = 1.1$, where $d(p_2, v_1) = d(p_2, v_3) = 1$, $v_3.\delta \in q.r$, and $v_1.\delta \notin q.r$.

Definition 7 *Top-k Spatiotemporal Semantic Place Retrieval Based on Temporal Range* Given a kSPT^r query q on a spatiotemporal RDF graph, the result of q includes k TQSPs minimizing scoring function $f(L^r(T_p),\ S(q,\ p))$, where $S(q,\ p)$ is the spatial distance between the query location and the root of the semantic place. Scoring function $f(L^r(T_p), S(q,\ p))$ is defined as

$$f(L^r(T_p), S(q,\ p)) = L^r(T_p) \times \hat{S}(q,\ p), \tag{4}$$

where $\hat{S}(q,\ p)$ is the normalized Euclidean distance defined in Eq. 1.

Recall that each place p has a unique TQSP, and therefore, it appears at most once in a kSPT^r result. The kSPT^r 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 based on the graph distance and belong to the temporal range of the query.

In the rest of the paper, to make the presentation and running examples concise, we ignore the normalization parameters L_τ, S_τ, D^t_τ in calculations.

3 Algorithms for kSP queries

3.1 Basic method: BSP

The most relevant existing work to our kSP queries are the top-k keyword queries on graphs [29,42]. Given a set of query keywords, the objective is to retrieve the top-k subtrees 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 w, the algorithm first determines the set of vertices whose documents contain w. 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 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 [26]. Like keyword search approaches, we also assume that the documents of the vertices in RDF graph are indexed by an inverted index [35]. 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 [65]), which enables efficient graph browsing operations (like BFS).

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 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 [31] 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 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 scores of all these places cannot be better than the current kth 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 Eq. 1; it can be easily adjusted if $f(L(T_p), S(q, p))$ is defined differently.

Example 6 Consider a 1SP query q located at $q.\lambda = q_1$ in Fig. 2 with query keywords $q.\psi = \{impressionist, art\}$, applying on the RDF graph of Fig. 1. Place p_1 is firstly retrieved from the R-tree and $L(T_{p_1}) = 3$ after calling function GETSEMANTICPLACE(). Therefore, the score of T_{p_1} is $f = 0.00042$. Next, T_{p_1} is added into H_k as the top-1 candidate and θ is updated to 0.042. Similarly, place p_2 is retrieved from the R-tree with $S(q_1, p_2) = 0.08 > \theta$. Then, T_{p_1} is returned as the top-1 result.

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

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

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 $q.\psi = \{impressionist, art\}$ as an example, the content of $M_{q.\psi}$ is shown in Table 2. Usually, the number of query keywords is small, and therefore, $M_{q.\psi}$ is small and cheap to construct.

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

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

v_1: {art}
v_2: {impressionist}
v_3: {impressionist}

path from p to every vertex in the posting list pl_i of keyword w_i, and then choose the vertex with the smallest shortest path distance. For example, in Fig. 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 w 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.

Example 7 Given query keywords $q.\psi = \{impressionist, art\}$, we illustrate function GETSEMANTICPLACE() (Algorithm 2) by constructing the TQSP for place p_1 in Fig. 1. BFS firstly reports p_1 that is added to T_{p_1}. However, $B \cap p_1.\psi_q = \emptyset$, which means that there is nothing to do for p_1. Next, v_1 is visited by BFS, we have $B \cap v_1.\psi_q = \{art\}$. Therefore, $L(T_{p_1}) = 1 + d(p, v_1) = 2$, and *art* is removed from B. Next, v_2 is visited, we have $B \cap v_2.\psi_q = \{impressionist\}$. Therefore, $L(T_{p_1}) = L(T_{p_1}) + d(p, v_2) = 3$, and *impressionist* is removed from B. Now, B becomes empty. Thus, $T_{p_1} = \langle p_1, (v_1, v_2) \rangle$ and $L(T_{p_1}) = 3$ are returned.

3.2 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 score of T_p is no less than threshold θ (the score of the k^{th} 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 semantic place retrieval with pruning algorithm (SPP).

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

```
1: T_p = ∅
2: L(T_p) = 1
3: Set B = q.ψ
4: while v = BFS(G, p) and B ≠ ∅ do
5:     Add v to T_p
6:     v.ψ_q = M_{q.ψ}.get(v)
7:     if B ∩ v.ψ_q ≠ ∅ then
8:         L(T_p) += |B ∩ v.ψ_q| × d(p, v)
9:         B = B \ v.ψ_q
10: if B ≠ ∅ then L(T_p) = +∞ and T_p = NULL
11: return L(T_p) and T_p
```

3.2.1 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 Fig. 1 and query keywords {*French*, *impressionist*}; no qualified semantic place rooted at p_2 exists, since p_2 never reaches *French*. Formally:

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

Testing whether p can reach a keyword w in the graph can be implemented by reachability queries [17,37,38,57,64] that have been well studied in the literature. TF-Label [17] 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 an independent component in our algorithm.

In the RDF graph, the documents of multiple vertices may share the same keyword w (as many as the length of the corresponding inverted list). For instance, in Fig. 1, the documents of v_2 and v_3 contain keyword *impressionist*.

Thus, in order to determine whether a place p can reach keyword *impressionist*, in the worst case three reachability queries (i.e., to v_2 and v_3) 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 w and added into the RDF graph. Edges are added from the vertices whose documents contain w to v_t. This way, for each query keyword w, it suffices to apply a single reachability query to v_t in order to find out whether any of the vertices whose documents contain w 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.

3.2.2 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* $q.\psi = \{w_1, \ldots, w_j, \ldots, w_m\}$*, 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* $L_B(T_p) = \sum_{i=1}^{j} d_g(p, w_i) + d(p, v) \times (m - j)$.

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., $d_g(p, w_n) \geq d(p, v)$, $j < n \leq m$. Therefore, we can have $L_B(T_p) = \sum_{i=1}^{j} d_g(p, w_i) + d(p, v) \times (m - j) \leq \sum_{i=1}^{m} d_g(p, w_i) = L(T_p)$. □

A TQSP T_p has a chance to be in the kSP result only if its score is less than threshold θ. Definition 8 presents the looseness threshold for all TQSPs that have not been computed yet.

Definition 8 *Looseness Threshold* Let θ be the 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 8, 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 score of T_p must be no smaller than the current kth candidate, then T_p cannot be in the result. □

By applying the two pruning rules, we can design algorithm SPP (Semantic Place search with Pruning), 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 8 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 = ∅
2: L_B(T_p) = 1
3: Set B = q.ψ
4: Compute the looseness threshold L_w(T_p)
5: while v = BFS(G, p) and B ≠ ∅ do
6:     Add v to T_p
7:     Compute the dynamic bound L_B(T_p)
8:     if L_B(T_p) ≥ L_w(T_p) then                    ▷ Pruning Rule 2
9:         return +∞ and T_p = NULL
10:    v.ψ = M_{q.ψ}.get(v)
11:    if B ∩ v.ψ ≠ ∅ then
12:        B = B \ v.ψ
13: if B ≠ ∅ then L(T_p) = +∞ and T_p = NULL
14: return L_B(T_p) and T_p
```

Example 8 Consider a kSP query q located at $q.\lambda = q_1$ in Fig. 2 with keywords $q.\psi = \{$*impressionist, art, galleries, museum, Paris*$\}$, requesting the top-1 TQSP in the RDF graph of Fig. 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 $6 \times 0.014 = 0.084$. Threshold θ is set to 0.084. 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 $L_w(T_{p_2}) = \theta / S(q_1, p_2) = 0.084/0.08 = 1.05$. 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. After v_3 is visited by BFS, $d(p_2, v_3) = 1$, and therefore, by Lemma 1, $L_B(T_{p_2})$ is increased by $|B| \times d(p_2, v_3) = 5$ and becomes 6. According to Pruning Rule 2, $L_B(T_{p_2}) = 6 > 1.05 = L_w(T_{p_2})$ 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.

3.3 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 scores, for pruning not only individual places but also sets of places, i.e., R-tree entries and the corresponding subtrees. We firstly introduce the α-radius word neighborhood in Definition 9, which is used for deriving the bounds.

Definition 9 *α-Radius word neighborhood of place* For place p, its α-radius word neighborhood $WN(p)$ contains the set of word–distance pairs $\{(w_i, d_g(p, w_i))\}$ where the shortest graph distance from p to each word w_i is no larger than α, i.e., $d_g(p, w_i) \leq \alpha$.

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 10.

Definition 10 *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 $\{(w_i, d_g(N, w_i)\}$ 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 w_i, $d_g(N, w_i) = \min_{p_j \in N} d_g(p_j, w_i)$. Obviously, $d_g(N, w_i) \leq \alpha$.

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 w appearing in v's document, if no corresponding pair for w is already in $WN(p)$, a new pair $(w, d(p, v))$ 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 $(w, d_g(e_i, w))$ in each $WN(e_i)$, if no corresponding pair for w is in $WN(N)$, $(w, d_g(e_i, w))$ is added to $WN(N)$ as $(w, d_g(N, w))$; otherwise, $d_g(N, w)$ is updated as $\min\{d_g(N, w), d_g(e_i, w)\}$.

Table 3 Example: 1-radius word neighborhoods

$q.\psi$	Museum	…	Art	French	Spanish
$d_g(p_1, w_i)$	1	…	1	1	–
$d_g(p_2, w_i)$	1	…	1	–	1
$d_g(N, w_i)$	1	…	1	1	1

Example 9 For $\alpha = 1$, part of the α-radius word neighborhoods of places p_1 and p_2 in Fig. 1 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.

Based on the α-radius word neighborhoods of places, we derive bounds of the looseness and the 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 $q.\psi = \{w_1, \ldots, w_j, \ldots, w_m\}$, 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 $L_B^\alpha(T_p) = \sum_{i=1}^{j} d_g(p, w_i) + (\alpha+1) \times (m-j)$ and $L_B^\alpha(T_p) \leq L(T_p)$.*

Lemma 3 α-Bound on the 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 score of T_p is $f_B^\alpha(p) = L_B^\alpha(T_p) \times S(q, p)$ and $f_B^\alpha(p) \leq f(L(T_p), S(q, p))$.*

Lemma 4 α-Bound on the looseness for nodes *Let $WN(N)$ be the α-radius word neighborhood of node N. Given query keywords $q.\psi = \{w_1, \ldots, w_j, \ldots, w_m\}$, 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 $L_B^\alpha(T_N) = \sum_{i=1}^{j} d_g(N, w_i) + (\alpha+1) \times (m-j)$ and $\forall p_i \in N$, $L_B^\alpha(T_N) \leq L(T_{p_i})$.

Lemma 5 α-Bound on the 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 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. $\forall p_i \in N$, $f_B^\alpha(N) \leq f(L(T_{p_i}), S(q, p_i))$.*

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 score of the kth candidate TQSP and $f_B^\alpha(p)$ be the α-bound on the score of the TQSP T_p rooted at p. If $f_B^\alpha(p) \geq \theta$, 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 score of the kth candidate TQSP and $f_B^\alpha(N)$ be the α-bound on the score of the TQSPs T_p rooted at places p enclosed in N. If $f_B^\alpha(N) \geq \theta$, 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 Fig. 1. For query keywords $q.\psi = \{$*French, art, museum*$\}$, based on Table 3 and Lemma 4, $L_B^\alpha(T_N) = 1 + 1 + 1 + 1 = 4$. Assuming the minimum spatial distance from N to a query location $q.\lambda$ is 2, by Lemma 5, $f_B^\alpha(N) = 8$. 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 Semantic Place 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 score rather than their spatial distance to the query location (lines 8 and 23), (ii) Pruning Rules 3 and 4 are used to discard places having no potential to be the result (line 22), and (iii) the termination condition is based on the α-bound on the 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 = ∅, ordered by f(L(T_p), S(q, p))
2: for each keyword w_i in q.ψ do
3:     Load posting list pl_i of w_i from I
4:     Load posting list of w_i from I^α
5: Construct M_{q.ψ}
6: θ = +∞
7: Queue Q = (root)
8: while e=GETNEXT(Q, R, q) do
9:     if f_B^α(e) ≥ θ then break
10:    if e refers to a place p then
11:        if e is Unqualified then                    ▷ Pruning Rule 1
12:            continue
13:        T_p = GETSEMANTICPLACEP(q.ψ, p, G, M_{q.ψ})
14:        if L(T_p) == +∞ then continue
15:        Compute the score f of T_p
16:        H_k.add(T_p, f)
17:        Update θ
18:    else                                            ▷ e refers to a node N
19:        for each entry e in N do
20:            Compute α-bound on the looseness L_B^α(T_e)
21:            Compute α-bound on the score f_B^α(e) for e
22:            if f_B^α(e) < θ then                    ▷ Pruning Rules 3 and 4
23:                Add (e, f_B^α(e)) to Q
24: return H_k
```

4 Algorithms for *k*SPT queries

This section presents the algorithms for the processing of kSPT^d and kSPT^r queries.

4.1 Processing *k*SPT^d queries

Section 4.1.1 explains how to store the temporal information in the inverted index, which helps to quickly find relevant vertices in the RDF graph, w.r.t., the query keywords and the query timestamps. Section 4.1.2 derives dynamic bounds on the temporal difference-based looseness (TDL). Section 4.1.3 presents α-radius temporal word neighborhoods and derives the α-bound on TDL. Section 4.1 introduces algorithm SPTD for the processing of kSPT^d queries. Four optimizations are proposed to improve the performance of SPTD in Sect. 4.1.5.

4.1.1 Temporal Inverted Index

The temporal inverted index I^t is the traditional inverted index with a temporal extension. The posting list of each term/word w is a list of pairs $(v, v.\delta)$, where v refers to a

vertex whose document contains w and $v.\delta$ is the timestamp associated with vertex v. The pairs in the posting lists are sorted in ascending order using the timestamp.

Let $d_m^t(q.\delta, w)$ be the minimum temporal difference between the query timestamp and the timestamps of the vertices whose documents contain term w, i.e.,

$$d_m^t(q.\delta, w) = \min_{v \in V \wedge w \in v.\psi} \{d^t(q.\delta, v.\delta)\}.$$

This can be used to derive a lower bound on the temporal difference-based looseness (TDL). The way of computing $d_m^t(q.\delta, w)$ is as follows. Given a kSPT^d query q, for each term w in $q.\psi$, its posting list is retrieved from the temporal inverted index I^t. Since the list is sorted, $d_m^t(q.\delta, w)$ can be quickly determined by one binary search operation and a few calculations.

4.1.2 Dynamic bound-based pruning

Lemma 6 Dynamic Bound on TDL *Given query keywords* $q.\psi = \{w_1, ..., w_j, .., w_m\}$, *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* $d(p, v)$. *Let* $V(w_i)$ *be the set of discovered vertices containing* w_i *and* $d_g^d(p, V(w_i)) = \min_{v_j \in V(w_i)}(1 + d(p, v_j)) \cdot (1 + d^t(q.\delta, v_j.\delta))$. *A lower bound* $L_B^d(T_p)$ *of the TDL is defined as*

$$\left(\sum_{i=1}^{j} \min\{d_g^d(p, V(w_i)), d_g^d(p, v, w_i)\} + \sum_{i=j+1}^{m} d_g^d(p, v, w_i) \right) / |q.\psi|, \quad (5)$$

where $d_g^d(p, v, w_i) = (1 + d(p, v)) \times (1 + d_m^t(q.\delta, w_i))$.

Proof The lower bound $L_B^d(T_p)$ consists of two parts. One part is for the query keywords that have not been discovered, i.e., $\sum_{i=j+1}^{m}(1 + d(p, v)) \times (1 + d_m^t(q.\delta, w_i))$. Obviously, the graph distance of any of these keywords must be no less than $d(p, v)$, since vertex v is the next vertex encountered in the BFS process. Also, $d_m^t(q.\delta, w_i)$ is the minimum temporal difference between the query timestamp and the timestamps of the vertices whose documents contain term w_i. Thus, for any keyword w_i in $\{w_{j+1}, \ldots, w_m\}$, $d_g^d(p, w_i) \geq (1+d(p, v)) \times (1+d_m^t(q.\delta, w_i))$. The other part is for the discovered query keywords, i.e., $\sum_{i=1}^{j} \min\{d_g^d(p, V(w_i)), (1 + d(p, v)) \times (1 + d_m^t(q.\delta, w_i))\}$. $d_g^d(p, V(w_i))$ is the minimum temporally weighted length of the path from p to the discovered vertices that contain w_i. However, beyond the discovered vertices, it is possible to have another vertex v' containing w_i with small temporal difference, so that the temporally weighted length of the path from p to v' is smaller than $d_g^d(p, V(w_i))$. But $(1 + d(p, v)) \times (1 + d_m^t(q.\delta, w_i)$ provides a lower bound on the temporally weighted length of the path from p to the undiscovered vertices containing w_i (the reason behind is similar to what is explained before). Hence, for any keyword w_i in $\{w_1, \ldots, w_j\}$, $d_g^d(p, w_i) \geq \min\{d_g^d(p, V(w_i)), (1 + d(p, v)) \times (1 + d_m^t(q.\delta, w_i)\}$. □

Pruning Rule 5 Dynamic Bound-Based Pruning using the TDL *According to Definition 8,* $L_w(T_p)$ *is the looseness threshold. For place* p, *as soon as* $L_B^d(T_p) \geq L_w(T_p)$, *the TQSP rooted at place* p *cannot be in the* kSPT^d *result, and thus* p *can be pruned.*

The proof is similar to that of Pruning Rule 2 and thus omitted.

4.1.3 α-Radius-based bounds

Definition 11 α-*Radius temporal word neighborhood of place* For place p, its α-radius temporal word neighborhood $WN^t(p)$ contains the set of word–distance–timestamp triples $\{(w_i, d(p, v), v.\delta)\}$ where v is a vertex containing word w_i (i.e., $w_i \in v.\psi$) and the shortest graph distance from p to v is no larger than α, i.e., $d(p, v) \leq \alpha$.

Based on the α-radius temporal word neighborhood of individual places, we define the α-radius temporal word neighborhoods of a set of places, i.e., a node in the R-tree, in Definition 12.

Definition 12 α-*Radius temporal word neighborhood of node* For a set of places $\{p_j\}$ enclosed in a node N of the R-tree, the α-radius temporal word neighborhood $WN^t(N)$ of N contains the set of word–distance–timestamp triples $\{w_i, d(p_j, v), v.\delta\}$ which is the union of the triples in $WN^t(p_j)$ of all places enclosed in N. Obviously, $d(p_j, v) \leq \alpha$.

The construction of α-radius temporal word neighborhood is similar to the construction of α-radius word neighborhood described in Sect. 3.3.

Based on the α-radius temporal word neighborhoods of places, we derive bounds of the TDL and the scores of TQSPs based on Lemmas 7 and 8. Lemmas 9 and 10 extend these bounds for sets of places rooted under R-tree nodes.

Lemma 7 α-Bound on the TDL of a place *Let* $WN^t(p)$ *be the* α*-radius temporal word neighborhood of place* p. *Given query keywords* $q.\psi = \{w_1, \ldots, w_j, \ldots, w_m\}$, *without loss of generality, assume that the first* j *keywords have corresponding triples in* $WN^t(p)$. *Let* $V(w_i)$ *be the set of vertices that appear in* $WN^t(p)$ *and contain* w_i. *The* α*-bound of the*

temporal difference-based looseness $L_B^{\alpha t}(T_p)$ *of TQSP* T_p *rooted at* p *is*

$$\left(\sum_{i=1}^{j} \min\left\{d_g^d(p, V(w_i)), (2+\alpha) \times (1+d_m^t(q.\delta, w_i))\right\} + \sum_{i=j+1}^{m} (2+\alpha) \times (1+d_m^t(q.\delta, w_i))\right) / |q.\psi|. \quad (6)$$

Lemma 8 α-Bound on the score for places *Let* $L_B^{\alpha t}(T_p)$ *be the* α*-bound on the temporal difference-based looseness of the TQSP* T_p *rooted at* p*. Given a kSPT*^{d} *query* q*, the* α*-bound on the score of* T_p *is* $f_B^{\alpha t}(p) = L_B^{\alpha t}(T_p) \times S(q, p)$ *and* $f_B^{\alpha t}(p) \leq f(L^d(T_p), S(q, p))$.

Lemma 9 α-Bound on the temporal difference-based looseness for nodes *Let* $WN^t(N)$ *be the* α*-radius temporal word neighborhood of node* N*. Given query keywords* $q.\psi = \{w_1, \ldots, w_j, \ldots, w_m\}$*, without loss of generality, assume that the first* j *keywords have corresponding triples in* $WN^t(N)$*. Let* $V(w_i)$ *be the set of vertices that appear in* $WN^t(N)$ *and contain* w_i*. The* α*-bound on the temporal difference-based looseness* $L_B^{\alpha t}(T_N)$ *of all the TQSPs* T_p *rooted at* p *enclosed in* N *is*

$$L_B^{\alpha t}(T_N) = \min_{p \in N} \left(\sum_{i=1}^{j} \min\left\{d_g^d(p, V(w_i)), (2+\alpha) \times (1+ \times d_m^t(q.\delta, w_i))\right\} + \sum_{i=j+1}^{m} (2+\alpha) \times (1+d_m^t(q.\delta, w_i))\right) / |q.\psi|. \quad (7)$$

Lemma 10 α-Bound on the score for nodes *Let* $L_B^{\alpha t}(T_N)$ *be the* α*-bound on the temporal difference-based looseness of the TQSPs* T_p *rooted at places* p *enclosed in* N*. Given a kSPT*^{d} *query* q*, the* α*-bound on the score of all the* T_p *rooted at* p *enclosed in* N *is* $f_B^{\alpha t}(N) = L_B^{\alpha t}(T_N) \times S(q, N)$*, where* $S(q, N)$ *is the minimum spatial distance between* q *and* N*.* $\forall p_i \in N$, $f_B^{\alpha t}(N) \leq f(L^d(T_{p_i}), S(q, p_i))$.

The proofs of Lemmas 7, 8, 9, and 10 are omitted for the interest of space. We proceed to introduce a pruning rule for places using Lemma 8 and a pruning rule for nodes using Lemma 10.

Pruning Rule 6 Place pruning *Given a kSPT*^{d} *query* q*, let* θ *be the score of the* k*th candidate TQSP and* $f_B^{\alpha t}(p)$ *be the* α*-bound on the score of the TQSP* T_p *rooted at* p*. If* $f_B^{\alpha t}(p) \geq \theta$*,* T_p *cannot be the kSPT*^{d} *result and* p *is pruned.*

Pruning Rule 7 R-tree node pruning *Given a kSPT*^{d} *query* q*, let* θ *be the score of the* k*th candidate TQSP and* $f_B^{\alpha t}(N)$ *be the* α*-bound on the score of the TQSPs* T_p *rooted at places* p *enclosed in* N*. If* $f_B^{\alpha t}(N) \geq \theta$*, the TQSP rooted at any place enclosed in* N *cannot be the result and* N *is pruned.*

4.1.4 SPTD algorithm

SPTD algorithm for the processing of kSPT^d queries is an extension of algorithm SP. It adopts the main routine of algorithm SP, but with the following modifications.

- The vertices having both documents and timestamps are indexed using the temporal inverted index I^t and the vertices with only documents are indexed using the inverted index I.
- The α-bound on the ranking score $f_B^{\alpha}(e)$ is updated to $f_B^{\alpha t}(e)$, where e could be either a place or a node.
- Pruning Rules 2, 3, and 4 are replaced by Pruning Rules 5, 6, and 7, respectively.

4.1.5 Optimizations

O1 In algorithm SPTD, $d_m^t(q.\delta, w)$ is the minimum temporal difference between the query timestamp and the timestamps of the vertices whose documents contain term w. The dynamic bound (Lemma 6) and the α-bounds (Lemmas 7 and 9) use $d_m^t(q.\delta, w)$ as the best possible temporal difference of either a place or a node in the R-tree. However, the vertex whose timestamp is used to compute $d_m^t(q.\delta, w)$ may not be reachable from the place and the node. Hence, the bounds derived based on $d_m^t(q.\delta, w)$ are not tight. We therefore consider the reachability to improve those bounds. Given a place p, let $d_m^t(q.\delta, w, p)$ be the minimum temporal difference between the query timestamp and the timestamps of the vertices whose documents contain term w and that are reachable from p, i.e.,

$$d_m^t(q.\delta, w, p) = \min_{v \in V \wedge w \in v.\psi \wedge p \sim v} \{d^t(q.\delta, v.\delta)\}.$$

Similarly, given a node N in the R-tree, we have

$$d_m^t(q.\delta, w, N) = \min_{v \in V \wedge w \in v.\psi \wedge N \sim v} \{d^t(q.\delta, v.\delta)\}.$$

Then, $d_m^t(q.\delta, w)$ in Lemmas 6 and 7 is replaced by $d_m^t(q.\delta, w, p)$. Similarly, $d_m^t(q.\delta, w)$ in Lemma 9 is replaced by $d_m^t(q.\delta, w, N)$. $d_m^t(q.\delta, w, p)$ is computed using the following steps.

1. Retrieve the posting list of w from the temporal inverted index.

2. Since the list is sorted, the closest timestamp of $q.\delta$ can be found by a binary search operation.
3. Check whether in the RDF graph, p reaches the vertex which has the closest timestamp using the TF-Label [17].
4. If it is unreachable, get the next closest timestamp and repeat step 3, until find a vertex v that is reachable from p.
5. $d_m^t(q.\delta, w, p) = d^t(q.\delta, v.\delta)$.

Regarding $d_m^t(q.\delta, w, N)$, where N encloses several places, the above method will issue too many reachability queries that are time-consuming. However, the number of nodes in the R-tree is small compared to the whole data set. Thus, we materialize the reachable vertices of each node and keep them in main memory. Given a node N, a vertex v is reachable from N if v is reachable from at least one place in N. On our experimental datasets, only megabytes are needed to store this data. Having the reachable vertices of each node, the above method can be used to compute $d_m^t(q.\delta, w, N)$ efficiently by replacing the TF-Label algorithm in step 3 with a simple check of whether the vertex who has the closest timestamp is in the reachable list of node N.

O2 Pruning Rule 1 used in algorithm SPTD has two drawbacks. Firstly, when determining whether a place can reach a word, in order to reduce the number of reachability queries, a vertex v_w is created for the word w and is added to the RDF graph. Edges between v_w and the vertices whose documents contain w are added. In this way, only one reachability query is enough. However, the cost of the reachability queries involving frequent words that connect many vertices is much higher than the cost of the reachability queries involving infrequent words. In order to improve the performance of algorithm SPTD, we store the reachable vertices for the words whose frequencies are larger than τ in main memory, denoted by R. Given a reachability query, if it involves a word whose frequency is larger than τ, the result will be immediately retrieved from R. Otherwise, TF-Label algorithm is called to compute the result. Large τ needs more computational time but less memory space, while small τ needs less computational time but more memory space. In the experiments, we have tuned parameter τ to achieve the best performance.

O3 The size of the α-radius temporal word neighborhood is larger than the size of the α-radius word neighborhood. This is because in the α-radius word neighborhood, only the minimum length of the shortest path from places/nodes to words are stored, while in the α-radius temporal word neighborhood, all the lengths of the shortest paths from places/nodes to words within the α-radius are stored, together with the corresponding timestamps. Due to the large size of the α-radius temporal word neighborhood, loading posting lists from index I^α is expensive, especially for high frequency words. Hence, we introduce a threshold l for the length of the posting lists stored in index I^α. During the construction of the α-radius temporal word neighborhood, the posting lists whose length is no greater than l are indexed in I^α. In the experiments, parameter l has been tuned to produce a good performance.

O4 In algorithm SPTD, Pruning Rule 1 only prunes unqualified places. We introduce Pruning Rule 8 that can prune unqualified nodes in the R-tree.

Pruning Rule 8 Unqualified Node Pruning *Let $p \nsim w$ denote that place p cannot reach keyword w in the RDF graph. Given query keywords $q.\psi$, node N is an unqualified node and can be pruned if $\forall p \in N \exists w \in q.\psi$, $p \nsim w$.*

The four optimizations improve the performance of algorithm SPTD, as shown in the experiments. Algorithm SPTD with the four optimizations is denoted by SPTD*.

4.2 Processing kSPT^r queries

This section derives dynamic bounds and α-bounds on the temporal range-based looseness (TRL). To evaluate kSPT^r queries, algorithm SPTR extends SPTD and algorithm SPTR* applies three of SPTD*'s optimizations.

Lemma 11 Dynamic Bound on TRL *Given query keywords $q.\psi = q.\psi^r \cup q.\psi^{\bar{r}}$, without loss of generality, suppose that we have already discovered a subset of keywords in $q.\psi^r$, denoted as $q.\psi_0^r$ and a subset of keywords in $q.\psi^{\bar{r}}$, denoted as $q.\psi_0^{\bar{r}}$ during the BFS exploration starting from p. Let $|q.\psi^r| - |q.\psi_0^r| = m$ and $|q.\psi^{\bar{r}}| - |q.\psi_0^{\bar{r}}| = n$. Let v be the next vertex encountered in the BFS process with graph distance $d(p, v)$. A lower bound of the TRL is defined as*

$$L_B^r(T_p) = 1 + \beta \cdot \frac{\min(L(T_p, q.\psi_0^r) + d(p, v) \cdot m, L_\tau)}{L_\tau} + (1-\beta) \cdot \frac{\min(L(T_p, q.\psi_0^{\bar{r}}) + d(p, v) \cdot n, L_\tau)}{L_\tau},$$

Lemma 12 α-Bound on the temporal range-based looseness of a place *Let $WN^t(p)$ be the α-radius temporal word neighborhood of place p. Given a kSPT^r query q with query keywords $q.\psi = q.\psi^r \cup q.\psi^{\bar{r}}$, let $WN^t(p, q.r)$ be a subset of $WN^t(p)$, such that*

$$\forall (w_i, d_g(p, v), v.\delta) \in WN^t(p, q.r), v.\delta \in q.r.$$

Without loss of generality, assume that a subset of keywords in $q.\psi^r$, denoted as $q.\psi_1^r$ and a subset of keywords in $q.\psi^{\bar{r}}$, denoted as $q.\psi_1^{\bar{r}}$ have corresponding triples in $WN^t(p, q.r)$. Let $|q.\psi^r| - |q.\psi_1^r| = m$ and $|q.\psi^{\bar{r}}| - |q.\psi_1^{\bar{r}}| = n$. The

α-bound of the temporal range-based looseness of TQSP T_p rooted at p is

$$L_B^{\alpha r}(T_p) = 1 + \beta \cdot \frac{\min(L(T_p, q.\psi_1^r) + d(p,v) \cdot m, L_\tau)}{L_\tau} + (1-\beta) \cdot \frac{\min(L(T_p, q.\psi_1^{\bar{r}}) + d(p,v) \cdot n, L_\tau)}{L_\tau}.$$

Lemma 13 α-Bound on the temporal range-based looseness for nodes *Let $WN^t(N)$ be the α-radius temporal word neighborhood of node N. Given a kSPT^r query q with query keywords $q.\psi = q.\psi^r \cup q.\psi^{\bar{r}}$, let $WN^t(N, q.r)$ be a subset of $WN^t(N)$, such that*

$$\forall (w_i, d_g(p,v), v.\delta) \in WN^t(N, q.r), v.\delta \in q.r.$$

Without loss of generality, assume that a subset of keywords in $q.\psi^r$, denoted as $q.\psi_2^r$ and a subset of keywords in $q.\psi^{\bar{r}}$, denoted as $q.\psi_2^{\bar{r}}$ have corresponding triples in $WN^t(N, q.\delta)$. The α-bound of the temporal range-based looseness of TQSP T_p rooted at p enclosed in N is

$$L_B^{\alpha r}(T_N) = \min_{p \in N}(1 + \beta \cdot \frac{\min(L(T_p, q.\psi_2^r) + d(p,v) \cdot m, L_\tau)}{L_\tau} + (1-\beta) \cdot \frac{\min(L(T_p, q.\psi_2^{\bar{r}}) + d(p,v) \cdot n, L_\tau)}{L_\tau}).$$

Algorithm SPTR for processing kSPT^r queries extends algorithm SPTD with the following modifications:

- TRL is used instead of TDL.
- The α-bound on the ranking score $f_B^{\alpha t}(e)$ is calculated based on $L_B^{\alpha r}(T_e)$ instead of $L_B^{\alpha t}(T_e)$, where e could be either a place or a node.
- The dynamic bound on TDL $L_B^d(T_p)$ used in Pruning Rule 5 is replaced by the dynamic bound on TRL $L_B^r(T_p)$.

Algorithm SPTR* further improves algorithm SPTR by adopting the following optimizations.

- Optimizations O2 and O3 introduced in Sect. 4.1.5 are applied.
- Pruning Rule 1 and Optimization O4 are extended by considering whether the reachable vertices fall within the query temporal range.

5 Experiments

This section evaluates the performance of the proposed algorithms, including (1) BSP (Sect. 3.1), SPP (Sect. 3.2), and SP (Sect. 3.3) for processing kSP queries, (2) SPTD and SPTD* (Sect. 4.1) for kSPT^d queries, and (3) SPTR and SPTR* (Sect. 4.2) for kSPT^r queries. We have conducted an empirical study using real datasets under various parameter settings.

5.1 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 and 1,138,751 vertices have timestamps. The number of distinct words in the documents of the vertices having timestamps is 955,904. 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 and 812,532 vertices have timestamps. The number of distinct words in the documents of the vertices having timestamps is 518,314. 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 ≥ 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 $[|q.\psi|/2, |q.\psi| \cdot \text{factor}]$ vertices, we randomly choose at most $|q.\psi|$ vertices, and $|q.\psi|$ keywords are randomly extracted from the distinct words in the documents of these vertices as the query keywords. We set factor $= 2$ which gives flexibility with respect to $|q.\psi|$

Table 4 Parameter settings

Parameter	Values
k	1, 3, **5**, 8, 10, 15, 20
$\|q.\psi\|$	1, 3, **5**, 8, 10
α	1, 2, **3**, 5
$\|q.r\|$	2, **6**, 14, 30, 60, 100, 200, 300

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. To generate $k\text{SPT}^d$ and $k\text{SPT}^r$ queries, we reuse the query locations and the keywords in the kSP queries. The query timestamps in the $k\text{SPT}^d$ queries are randomly selected from the timestamps of those chosen $|q.\psi|$ vertices. The query temporal ranges in the $k\text{SPT}^r$ queries are generated by taking the selected query timestamps as centers and expanding a small range.

Parameter settings Performance is evaluated by varying the number of requested results k, the number of query keywords $|q.\psi|$, α of the α-radius-based bounds, the size of the query temporal range $|q.r|$, and also the data size for scalability evaluation. We vary one parameter while fixing the others. Table 4 lists the values of the parameters. The values in bold are the default values for the parameters. 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, the inverted indexes, and the temporal inverted indexes are shown in Table 5. 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. The temporal inverted index contains the documents with timestamps, which refer to the vertices having both documents and timestamps in the RDF graph. Note that some of the vertices with documents do not have timestamps. Hence the size of the temporal inverted index is smaller than the size of the inverted index.

5.2 Efficiency evaluation of *k*SP queries

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 GETSEMANTICPLACE 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.

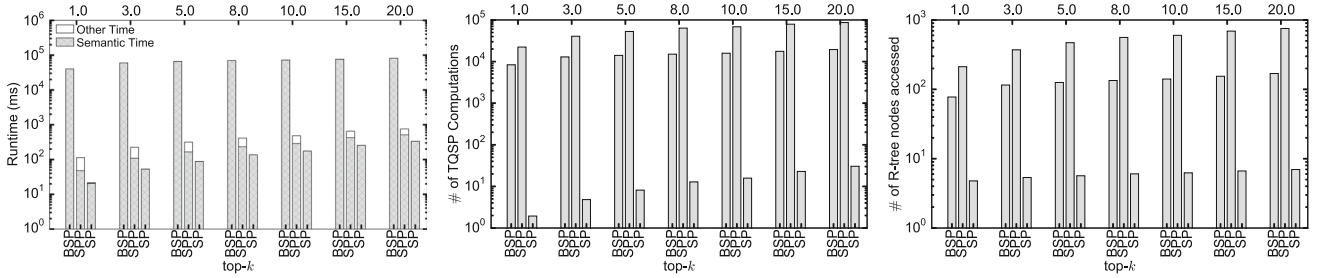
Varying k Figures 3 and 4 show the cost of all methods on dataset DBpedia and Yago, respectively. As expected, the runtime, the number of TQSP computations, and the number of R-tree nodes accessed all increase as k increases, since a larger number of requested semantic places requires exploring a larger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

On dataset DBpedia (Fig. 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 Fig. 3a). SPP includes other costs (i.e., "other time" in Fig. 3a), 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 Fig. 3b 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 (Fig. 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 Sect. 3.2 reduce the cost for TQSP computations significantly, but at the expense of performing reachability queries (i.e., the "other time" in Fig. 4a). 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.

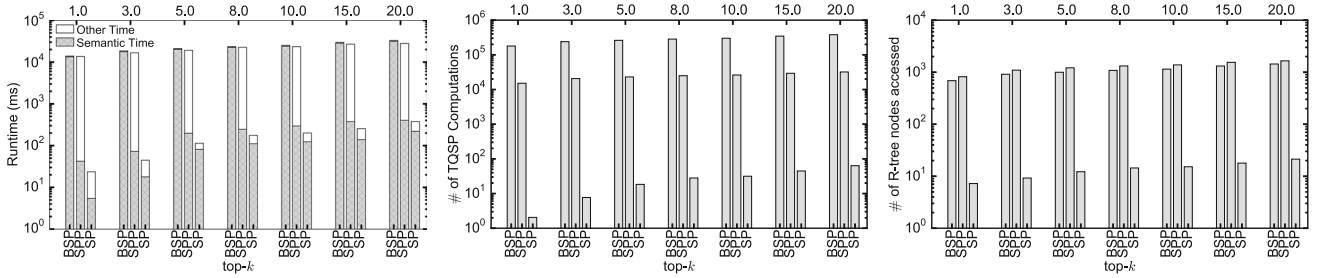
Table 5 Storage cost

Data	R-tree (MB)	RDF graph (MB)	Inverted Index (MB)	Temporal Inverted Index (MB)
DBpedia	50.54	607.95	1307.98	118
Yago	273.17	454.81	231.91	22



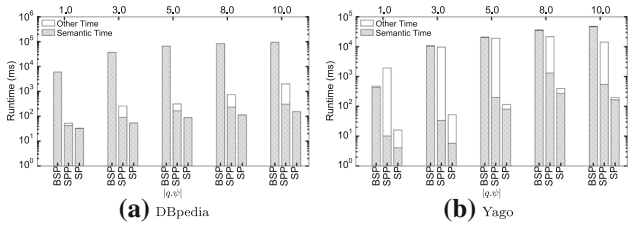
(a) runtime **(b)** number of TQSP computations **(c)** number of R-tree nodes accessed

Fig. 3 Varying k on DBpedia (kSP)



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

Fig. 4 Varying k on Yago (kSP)



(a) DBpedia **(b)** Yago

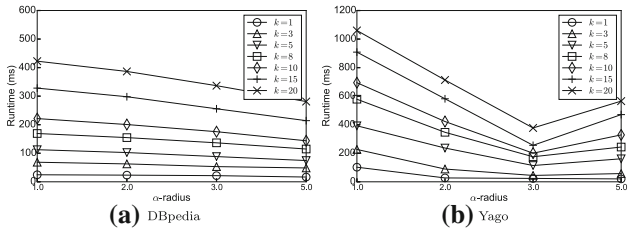
Fig. 5 Varying $|q.\psi|$ (kSP)

Table 6 α-Radius word neighborhood size

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

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.

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 204.70 GB on DBpedia and 30.63 GB on Yago when $\alpha = 5$.



(a) DBpedia **(b)** Yago

Fig. 6 Varying α (kSP)

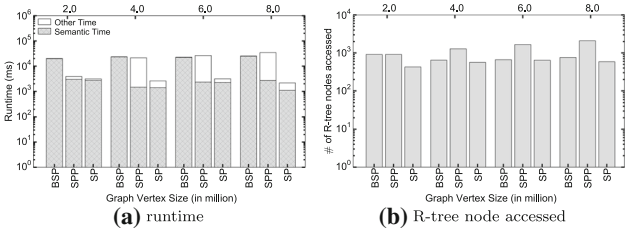
We evaluate the performance of SP with $k = 1, 3, 5, 8, 10, 15, 20$ when varying α from 1 to 5 on DBpedia and Yago (Fig. 6). The number of query keywords is fixed to $|q.\psi| = 5$. Note that with a larger α, the exploring direction of SP is more biased to TQSP looseness than spatial distance (Lemmas 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 Fig. 6b. 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).

Scalability In this section, we evaluate the performance of the three methods on datasets of different sizes. We adopt the random jump sampling method [43] 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

Table 7 Datasets extracted from Yago

# 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



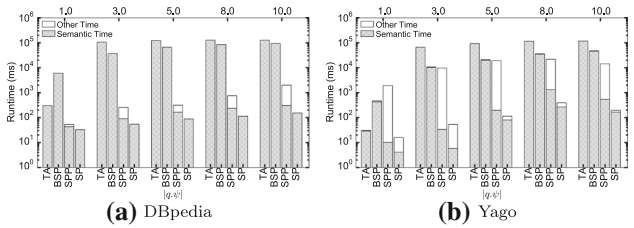
(a) runtime **(b)** R-tree node accessed

Fig. 7 Varying graph size by sampling (Yago)

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.

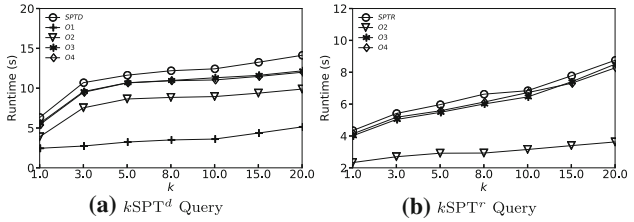
Comparison with top-k aggregation BSP and its optimized versions (SPP and SP) examine places in increasing spatial distance from the query location and compute their looseness as necessary, until the top-k places are confirmed. It is also possible to evaluate kSP queries by a hybrid approach that combines 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 [42] (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 [22]: 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 Fig. 8 for queries with various numbers of keywords $|q.\psi|$. On DBpedia, only when $|q.\psi| = 1$ TA performs better than BSP while being eight 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



(a) DBpedia **(b)** Yago

Fig. 8 Comparison with top-k aggregation (TA)



(a) kSPT^d Query **(b)** kSPT^r Query

Fig. 9 Optimization evaluation on Yago

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.

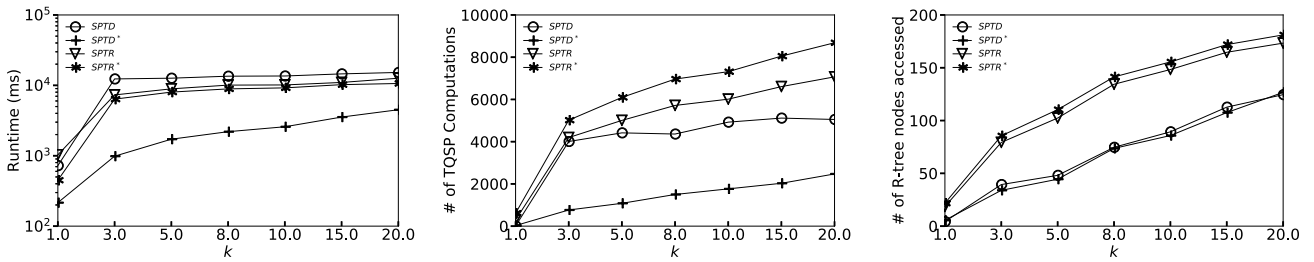
In summary, while it is extremely cheap to compute spatial distances and conduct spatial nearest neighbor search, it is expensive to conduct graph browsing and incremental ranking of places by looseness. This imbalance between the costs of computing spatial distance and looseness motivated the design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

5.3 Efficiency evaluation of *k*SPT queries

Optimization evaluation The proposed optimization techniques are evaluated separately in this experiment. Optimizations O1, O2, O3, and O4 are applied for kSPT^d queries and optimizations O2, O3, and O4 are applied for kSPT^r queries. Figure 9 shows the runtime after applying each of the optimizations to the kSPT^d and kSPT^r queries. The result shows that the runtime is progressively reduced by each optimization in algorithms SPTD and SPTR, which justifies their value. In the rest of the experiments, we show the performance of algorithms SPTD* and SPTR* that include all optimizations.

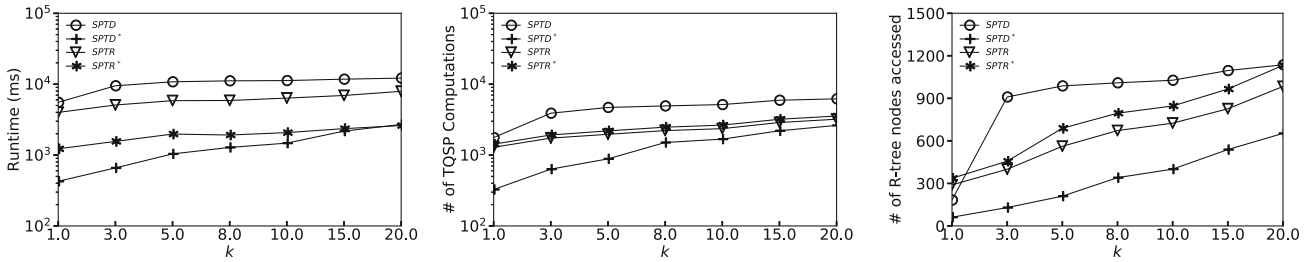
Varying k Figures 10 and 11 show the cost of algorithms SPTD, SPTD*, SPTR, and SPTR* on datasets DBpedia and Yago, respectively. As k increases, the runtime, the number of TQSP computations, and the number of accessed R-tree nodes of all the algorithms increase, which is expected. Because all the algorithms find results in the search space in an incremental fashion. The more results are requested, the more the computational effort to explore the search space is. Algorithms SPTD* and SPTR* outperform algorithms SPTD and SPTR, respectively. This is because the optimizations proposed in Sect. 4.1.5 take effect in reducing the computational cost.

Varying $|q.\psi|$ Figure 12 compares the runtimes of algorithms SPTD, SPTD*, SPTR, and SPTR* on datasets DBpedia and Yago. The results show that algorithms SPTD* and SPTR* are more efficiently than algorithms SPTD and SPTR. The runtimes of the algorithms slightly increase as the number of keywords increases on DBpedia. The reason is that more vertices in the RDF graph will be explored to discover TQSPs covering all the query keywords in $|q.\psi|$. On Yago, the runtimes of the algorithms are not sensitive to the number of query keywords. The reason is that Yago is smaller in size than DBpedia, so that computing a single TQSP on Yago is faster. Thus, the time spent on exploring the vertices to match more query keywords is not much. Then the overall runtime is almost stable as $|q.\psi|$ increases.



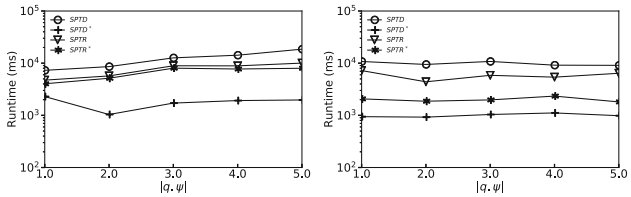
(a) runtime **(b)** number of TQSP computations **(c)** number of R-tree nodes accessed

Fig. 10 Varying k on DBpedia (kSPT)



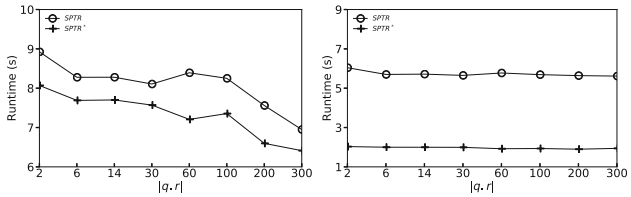
(a) runtime **(b)** number of TQSP computations **(c)** number of R-tree nodes accessed

Fig. 11 Varying k on Yago (kSPT)



(a) DBpedia **(b)** Yago

Fig. 12 Varying $|q.\psi|$ (kSPT)



(a) DBpedia **(b)** Yago

Fig. 13 Varying $|q.r|$ (kSPT)

Varying $|q.r|$ Figure 13 shows the runtimes of algorithms SPTR and SPTR* when varying the temporal range $|q.r|$ in $k\text{SPT}^r$ queries. On dataset DBpedia, the runtime decreases as $|q.r|$ increases, while on dataset Yago, the runtime is almost stable. This is because on DBpedia computing a single TQSP is expensive and more TQSPs are calculated when $|q.r|$ is small, compared with the case when $|q.r|$ is large. However, on Yago, although the number of TQSP computations is also larger when $|q.r|$ is small, computing a single TQSP is fast, so that the overall runtime does not change significantly with $|q.r|$. This experiment again shows that SPTR* outperforms SPTR.

Tuning α **and the length of the posting lists** l In this experiment, we evaluate the effect of parameter α and l in algorithms SPTD* and SPTR*. Table 8 displays the total space that the α-radius temporal word neighborhoods occupy, for the two datasets and different values of α and l. As expected, the space increases with both α and l.

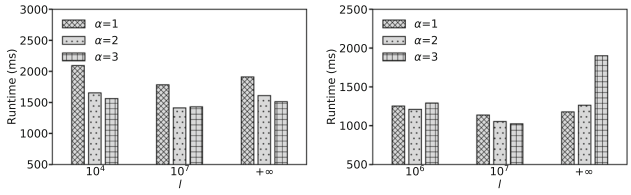
Figures 14 and 15 show the performance of algorithms SPTD* and SPTR* when varying α from 1 to 3 and varying l (parameter in optimization O3) from 10^4 to $+\infty$ on DBpedia and Yago. We observe the following effects of the two parameters. Firstly, large α values enable tighter bound derivations and facilitate the pruning of more pruned places and nodes, so that the computational cost can be reduced. However, large α will produce a big index, so that the time needed for loading data from disk increases. When the computational cost saved by large α does not compensate the cost incurred by loading data, the overall runtime increases. Thus, there is a tradeoff between the saving for computation and cost for transferring data brought by α. Secondly, when fixing α, a small l will not incur high cost of data loading, but make the bounds loose so that the computational cost is high. On the contrary, large l values may help to derive tight bounds to reduce computational cost, but incur high cost for data loading. Overall, based on the empirical evaluation of different α and l values on the two datasets, we recommend $\alpha = 3, l = 10^7$ for both Yago and DBpedia w.r.t., both performance gains and the α-radius temporal word neighborhood size (i.e., index size).

Tuning τ Figure 16 shows the runtimes of algorithms SPTD* and SPTR* when varying τ (parameter in optimization O2) on DBpedia and Yago. Table 9 shows the size of the α-radius temporal word neighborhood index I^α for different values of τ. As expected, as τ increases, more data will be stored in the index so that more storage is needed. The runtime of algorithms SPTD* and SPTR* on Yago firstly slightly increases and then increases significantly as τ increases. Algorithm SPTR* exhibits an increasing trend on DBpedia. This result shows that optimization O2 does improve the performance. However, algorithm SPTD* is not sensitive to τ on DBpedia. By taking both the runtime and the storage into account, we recommend $\tau = 10^4$ for DBpedia and $\tau = 10^3$ for Yago.

Scalability This experiment evaluates the performance of algorithms SPTD, SPTD*, SPTR, and SPTR* on datasets of different sizes. We reuse the four datasets generated for kSP queries. Figure 17 shows the performance of the four algo-

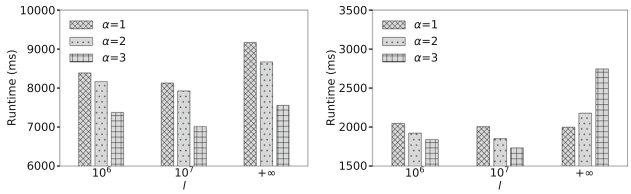
Table 8 α-Radius temporal word neighborhood size

	DBpedia			Yago		
	$\alpha = 1$	$\alpha = 2$	$\alpha = 3$	$\alpha = 1$	$\alpha = 2$	$\alpha = 3$
$l = 10^4$	45 MB	81 MB	118 MB	15 MB	20 MB	20 MB
$l = 10^6$	321 MB	1.1 MB	2.8 GB	140 MB	856 MB	1.8 GB
$l = 10^7$	1.1 GB	3.1 GB	7.8 GB	170 MB	1.3 GB	4.9 GB
$l = \infty$	1.1 GB	3.9 GB	15 GB	170 MB	1.4 GB	5.6 GB



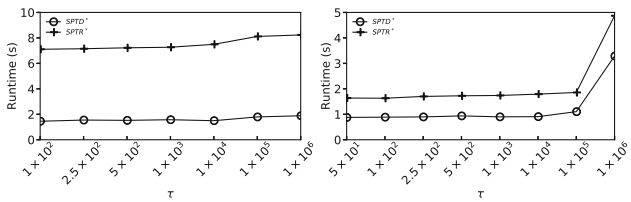
(a) DBpedia **(b)** Yago

Fig. 14 Tuning α and l for $k\text{SPT}^d$ Queries



(a) DBpedia **(b)** Yago

Fig. 15 Tuning α and l for $k\text{SPT}^r$ Queries



(a) DBpedia **(b)** Yago

Fig. 16 Varying τ (kSPT)

rithms 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. Algorithms SPTD* and SPTR* outperform algorithms SPTD and SPTR and the runtimes of all four algorithms increase in a linear fashion.

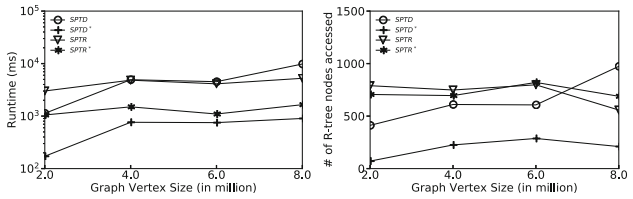
6 Related work

Keyword search on graph data Due to its user-friendly query interface, keyword search is not only the de facto information retrieval method for WWW data but also a popular querying mechanism for XML documents [18,25], relational databases [12,34], and graph data [19,29,39]. Traditional graph search algorithms convert queries into search over feature spaces, such as paths [54], frequent-patterns [63], and sequences [36], which focus more on the structure of the graph rather than the semantic content of the graph. Nevertheless, keyword search over graph data [12,18,19,25,29,34,39] 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 reinforcement 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 [61]. Liu et al. [48] study the keyword-based search on temporal graphs with optional predicates and ranking functions related to timestamps. Zhong et al. [66] search for a both locally and globally diverse set of most relevant results for a given keyword query on graphs.

Keyword search on RDF data RDF data are a special type of graph data, traditionally queried using structured query languages, like SPARQL. There has been increasing interest in keyword queries over RDF data. SPARQL queries are augmented with keywords for ranked retrieval of RDF data [21]. A keyword-based retrieval model over RDF graphs [20]

Table 9 Size of α-radius temporal word neighborhood index I^α

τ	50	100	250	500	10^3	10^4	10^5	10^6
Yago								
Storage	6.1G	4.0G	2.0G	942M	455M	32M	3.2M	332K
τ	–	100	250	500	10^3	10^4	10^5	10^6
DBpedia								
Storage	–	58G	31G	19G	12G	1.7G	53M	332K



(a) Yago **(b)** Yago

Fig. 17 Varying graph size by sampling

identifies a set of maximal subgraphs whose vertices contain the query keywords. These subgraphs are ranked based on statistical language models (LMs) [52]. Top-k exploration of query candidates over RDF [56] 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 [42]. Both [42,56] follow the definition of BLINKS [29] for the result subgraphs. k-nearest keyword (k-NK) search on RDF graphs [45] finds the k closest pairs of vertices, (v_i, u_i) that contain two given keywords q and w, respectively. SemDIS [27] demonstrates a system, where semantic associations are discovered in a large semantic metabase represented in RDF. Keyword query interpretation [23] 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 [24] 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 [14] 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 [16] captures associations across RDF paths for improving the query execution performance. Recently, a query graph assembly approach [28] converts keyword queries into graph queries. SPARQL and keyword search has been integrated to find SPARQL matches that are closest to all keywords in RDF graphs [51]. A type-based summary which summarizes all the inter-entity relationships from RDF data is used for keywords-to-SPARQL translation [47].

Spatial RDF stores Recently, there are many efforts toward the efficient storage and indexing of spatial RDF data. Parliament [7] is an implementation of GeoSPARQL. Strabon [41] 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. [15] 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 [58] uses a Hilbert space-filling curve to index the space and supports spatial range queries and NN search. S-Store [59] 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 [10], and OWLIM-SE [6], support spatial RDF data management, however, details about their internal design are not available. A spatial encoding scheme for RDF stores that supports efficient spatial data management was proposed in Liagouris et al. [44]. 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 and kSPT 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 and kSPT queries. In fact, extensions are expected to be highly inefficient because they do not guide search based on the spatial and the temporal arguments in the query. Thirdly, kSP and kSPT queries enable users to search nearby places that semantically and temporally match their preferences (expressed by keywords and temporal arguments). Such a functionality finds important and useful applications as discussed in Sect. 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 and kSPT search on very large RDF data.

7 Conclusion

In this paper, we proposed a top-k relevant semantic place retrieval (kSP) query and a top-k spatiotemporal semantic place retrieval (kSPT) query. The kSP query 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. The kSPT query extends the kSP query by incorporating the temporal dimension. Both the two types of queries do not require the use of any structured query languages, and 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 algo-

rithm (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 Sect. 3.3, 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 algorithms for kSP queries have been extended and improved for evaluating kSPT queries. 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 and kSPT queries efficiently 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 problem definition follows directly from previous work of RDF keyword search [23,29,42,56], 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).

References

1. Alternative fueling station locator. http://www.afdc.energy.gov/locator/stations/
2. Crime in chicagoland. http://crime.chicagotribune.com/
3. Data.gov. http://www.data.gov
4. Dbpedia. http://wiki.dbpedia.org
5. Hospital compare. http://health.data.gov/def/cqld
6. Owlim-se. http://owlim.ontotext.com/display/OWLIMv43/OWLIM-SE
7. Parliament. http://parliament.semwebcentral.org
8. Patients like me. www.patientslikeme.com
9. Spot crime. http://www.spotcrime.com/
10. Virtuoso. http://virtuoso.openlinksw.com
11. Yago. http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/
12. Agrawal, S., Chaudhuri, S., Das, G.: Dbxplorer: a system for keyword-based search over relational databases. In: ICDE, pp. 5–16 (2002)
13. Battle, R., Kolas, D.: Enabling the geospatial semantic web with parliament and geosparql. Semant. Web **3**(4), 355–370 (2012)
14. Bikakis, N., Giannopoulos, G., Liagouris, J., Skoutas, D., Dalamagas, T., Sellis, T.: Rdivf: diversifying keyword search on RDF graphs. In: TPDL, pp. 413–416 (2013)
15. Brodt, A., Nicklas, D., Mitschang, B.: Deep integration of spatial query processing into native RDF triple stores. In: SIGSPATIAL, pp. 33–42 (2010)
16. Cappellari, P., Virgilio, R.D., Maccioni, A., Roantree, M.: A path-oriented RDF index for keyword search query processing. In: DEXA, pp. 366–380 (2011)
17. Cheng, J., Huang, S., Wu, H., Fu, A.W.: Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In: SIGMOD, pp. 193–204 (2013)
18. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: Xsearch: a semantic search engine for XML. In: VLDB, pp. 45–56 (2003)
19. Dalvi, B.B., Kshirsagar, M., Sudarshan, S.: Keyword search on external memory data graphs. PVLDB **1**(1), 1189–1204 (2008)
20. Elbassuoni, S., Blanco, R.: Keyword search over RDF graphs. In: CIKM, pp. 237–242 (2011)
21. Elbassuoni, S., Ramanath, M., Schenkel, R., Weikum, G.: Searching RDF graphs with SPARQL and keywords. IEEE Data Eng. Bull. **33**(1), 16–24 (2010)
22. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: PODS (2001)
23. Fu, H., Anyanwu, K.: Effectively interpreting keyword queries on RDF databases with a rear view. In: ISWC, pp. 193–208 (2011)
24. Giannopoulos, G., Biliri, E., Sellis, T.: Personalizing keyword search on RDF data. In: TPDL, pp. 272–278 (2013)
25. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: ranked keyword search over XML documents. In: SIGMOD, pp. 16–27 (2003)
26. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD, pp. 47–57 (1984)
27. Halaschek-Wiener, C., Aleman-Meza, B., Arpinar, I.B., Sheth, A.P.: Discovering and ranking semantic associations over a large RDF metabase. In: VLDB, pp. 1317–1320 (2004)
28. Han, S., Zou, L., Yu, J.X., Zhao, D.: Keyword search on RDF graphs: a query graph assembly approach. In: CIKM, pp. 227–236 (2017)
29. He, H., Wang, H., Yang, J., Yu, P.S.: BLINKS: ranked keyword searches on graphs. In: SIGMOD, pp. 305–316 (2007)
30. Hendler, J.A., Holm, J., Musialek, C., Thomas, G.: US government linked open data: semantic.data.gov. IEEE Intell. Syst. **27**(3), 25–31 (2012)
31. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Trans. Database Syst. **24**(2), 265–318 (1999)
32. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: YAGO2: a spatially and temporally enhanced knowledge base from Wikipedia. Artif. Intell. **194**, 28–61 (2013)
33. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient IR-style keyword search over relational databases. In: VLDB, pp. 850–861 (2003)
34. Hristidis, V., Papakonstantinou, Y.: DISCOVER: keyword search in relational databases. In: VLDB, pp. 670–681 (2002)
35. Inglis, J.: Inverted indexes and multi-list structures. Comput. J. **17**(1), 59–63 (1974)
36. Jiang, H., Wang, H., Yu, P.S., Zhou, S.: Gstring: a novel approach for efficient search in graph databases. In: ICDE, pp. 566–575 (2007)
37. Jin, R., Ruan, N., Dey, S., Yu, J.X.: SCARAB: scaling reachability computation on large graphs. In: SIGMOD, pp. 169–180 (2012)
38. Jin, R., Ruan, N., Xiang, Y., Wang, H.: Path-tree: an efficient reachability indexing scheme for large directed graphs. ACM Trans. Database Syst. **36**(1), 7 (2011)
39. Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H.: Bidirectional expansion for keyword search on graph databases. In: VLDB, pp. 505–516 (2005)
40. Koubarakis, M., Kyzirakos, K.: Modeling and querying metadata in the semantic sensor web: the model stRDF and the query language stSPARQL. In: ESWC, pp. 425–439 (2010)
41. Kyzirakos, K., Karpathiotakis, M., Koubarakis, M.: Strabon: a semantic geospatial DBMS. In: ISWC, pp. 295–311 (2012)

42. Le, W., Li, F., Kementsietsidis, A., Duan, S.: Scalable keyword search on large RDF data. TKDE **26**(11), 2774–2788 (2014)
43. Leskovec, J., Faloutsos, C.: Sampling from large graphs. In: KDD, pp. 631–636 (2006)
44. Liagouris, J., Mamoulis, N., Bouros, P., Terrovitis, M.: An effective encoding scheme for spatial RDF data. PVLDB **7**(12), 1271–1282 (2014)
45. Lian, X., Hoyos, E.D., Chebotko, A., Fu, B., Reilly, C.: K-nearest keyword search in RDF graphs. J. Web Sem. **22**, 40–56 (2013)
46. Libkin, L., Reutter, J.L., Soto, A., Vrgoc, D.: Trial: a navigational algebra for RDF triplestores. ACM Trans. Database Syst. **43**(1), 5:1–5:46 (2018)
47. Lin, X., Ma, Z., Yan, L.: RDF keyword search using a type-based summary. J. Inf. Sci. Eng. **34**(2), 489–504 (2018)
48. Liu, Z., Wang, C., Chen, Y.: Keyword search on temporal graphs. In: ICDE, pp. 1807–1808 (2018)
49. Neumann, T., Weikum, G.: RDF-3X: a risc-style engine for RDF. PVLDB **1**(1), 647–659 (2008)
50. Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.: Query processing in spatial network databases. In: VLDB, pp. 802–813 (2003)
51. Peng, P., Zou, L., Qin, Z.: Answering top-k query combined keywords and structural queries on RDF graphs. Inf. Syst. **67**, 19–35 (2017)
52. Ponte, J.M., Croft, W.B.: A language modeling approach to information retrieval. In: SIGIR, pp. 275–281 (1998)
53. Prud'Hommeaux, E., Seaborne, A., et al.: Sparql query language for rdf. W3C Recomm. **15** (2008). https://www.w3.org/TR/rdfsparql-query
54. Shasha, D., Wang, J.T.L., Giugno, R.: Algorithmics and applications of tree and graph searching. In: PODS, pp. 39–52 (2002)
55. Shi, J., Wu, D., Mamoulis, N.: Top-k relevant semantic place retrieval on spatial RDF data. In: SIGMOD, pp. 1977–1990 (2016)
56. Tran, T., Wang, H., Rudolph, S., Cimiano, P.: Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In: ICDE, pp. 405–416 (2009)
57. van Schaik, S.J., de Moor, O.: A memory efficient reachability data structure through bit vector compression. In: SIGMOD, pp. 913–924 (2011)
58. Wang, C., Ku, W., Chen, H.: Geo-store: a spatially-augmented SPARQL query evaluation system. In: SIGSPATIAL, pp. 562–565 (2012)
59. Wang, D., Zou, L., Feng, Y., Shen, X., Tian, J., Zhao, D.: S-store: an engine for large RDF graph integrating spatial information. In: DASFAA, pp. 31–47 (2013)
60. Wang, D., Zou, L., Zhao, D.: GST-store: an engine for large RDF graph integrating spatiotemporal information. In: EDBT, pp. 652–655 (2014)
61. Wang, H., Aggarwal, C.C.: A survey of algorithms for keyword search on graph data. In: Managing and Mining Graph Data, pp. 249–273 (2010)
62. Wylot, M., Hauswirth, M., Cudré-Mauroux, P., Sakr, S.: RDF data storage and query processing schemes: a survey. ACM Comput. Surv. **51**(4), 84:1–84:36 (2018)
63. Yan, X., Yu, P.S., Han, J.: Substructure similarity search in graph databases. In: SIGMOD, pp. 766–777 (2005)
64. Yildirim, H., Chaoji, V., Zaki, M.J.: GRAIL: scalable reachability index for large graphs. PVLDB **3**(1), 276–284 (2010)
65. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale RDF data. PVLDB **6**(4), 265–276 (2013)
66. Zhong, M., Wang, Y., Zhu, Y.: Coverage-oriented diversification of keyword search results on graphs. In: DASFAA, pp. 166–183 (2018)
67. Zou, L., Mo, J., Chen, L., Özsu, M.T., Zhao, D.: gStore: answering SPARQL queries via subgraph matching. PVLDB **4**(8), 482–493 (2011)