K-SPIN: Efficiently Processing Spatial Keyword Queries on Road Networks

Tenindra Abeywickrama, Muhammad Aamir Cheema, Arijit Khan

Abstract—A significant proportion of all search volume consists of local searches. As a result, search engines must be capable of finding relevant results combining both spatial proximity and textual relevance with high query throughput. We observe that existing techniques answering these *spatial keyword queries* use keyword aggregated indexing, which has several disadvantages on road networks. We propose K-SPIN, a versatile framework that instead uses *keyword separated indexing* to delay and avoid expensive operations. At first glance, this strategy appears to have impractical pre-processing costs. However, by exploiting several useful observations, we make the indexing cost not only viable but also light-weight. For example, we propose a novel ρ-Approximate Network Voronoi Diagram (NVD) with one order of magnitude less space cost than exact NVDs. By carefully exploiting features of the K-SPIN framework, our query algorithms are up to two orders of magnitude more efficient than the state-of-the-art as shown in our experimental investigation on various queries, parameter settings, and real road network and keyword datasets.

Index Terms—Road networks, points of interest search, spatio-textual queries, network Voronoi diagrams

1 Introduction

Finding the nearest relevant points of interest (POIs) to a user's location is among the most popular queries in map-based services [1]. These POIs are often associated with rich textual descriptions in addition to their spatial locations. Consider the example road network in Figure 1 with unit edge weights and 8 objects (POIs) each associated with a set of keywords. A *spatial keyword* query retrieves objects that are close to the query location (e.g., in terms of travel time over road networks) and are textually relevant. The following two types of spatial keyword queries have been studied on road networks.

Boolean kNN Query [2]. Given a set of query keywords, a Boolean kNN (BkNN) query returns the k objects closest to the query location among those that satisfy the keyword criteria. The criteria may be disjunctive (contain *any* query keyword) or conjunctive (contain *all* query keywords). For example, a user may want to find the closest object that contains either "restaurant" *or* "takeaway". In Figure 1, the answer is o_8 because no object closer to query location q contains either "restaurant" or "takeaway". Another user may wish to find the closest POI containing both "Thai" *and* "restaurant". In Figure 1, the result would then be o_6.

Top-k Spatial Keyword Query [3], [4]. A top-k spatial keyword query returns k objects with the best scores. The score of an object is computed using a function combining the object's network distance from q and the relevance of the object's textual description with the query keywords. Section 2 provides a formal description.

20 billion Google searches with a location component are performed every quarter (including 13.9 billion from mobile devices) [5]. This translates to ≈2500 spatial keyword queries per second on average. Using network distance affords greater accuracy and flexibility (e.g., using travel-time rather than the distance "as-the-crow-flies"). But efficiently indexing road networks and keyword information to meet such high throughput demands is a challenging problem. Moreover, the indexing strategy used by current road network spatial keyword techniques, called keyword aggregation, leaves substantial room for improvement.

- *T. Abeywickrama and M. A. Cheema are with the Faculty of Information Technology, Monash University, Australia. E-mail: {tenindra.abeywickrama,aamir.cheema}@monash.edu*
- *A. Khan is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. E-mail: arijit.khan@ntu.edu.sg*

1.1 Motivation

Keyword aggregation is the idea of summarizing keyword occurrences over geographical regions. Spatial keyword queries are then answered by searching the most promising regions first, while pruning regions that cannot contain results. This technique is used extensively by spatial keyword query techniques in Euclidean space [6], [7], [8], [9]. Notably, *all* existing techniques for road networks also use the idea of keyword aggregation. The disadvantage of keyword aggregation is the generation of many false positives. Whenever a candidate is encountered, its distance from the query must be computed to confirm if it is relevant or not. Computing distance in Euclidean space is a quick arithmetic operation, but in road networks computing distance is a complex graph operation and far more expensive. Consequently the penalty paid for incurring false positives in road networks is significantly higher than in Euclidean space. So, while keyword aggregation is useful for Euclidean space, it is far less effective for road networks. We illustrate this problem in an example using a state-of-the-art spatial keyword technique for road networks [4].

Consider again the objects and road network with unit edge weights shown in Figure 1. The existing techniques first groups objects, e.g., G-tree [4] may form four groups G_1, G_2, G_3, and G_4 (shown by rectangles with broken lines). Keywords are then aggregated by creating a pseudo-document for each group that is the union of keyword occurrences over all the contained objects. For example, G_1's pseudo-document contains keywords "Italian", "restaurant", "takeaway", "Thai", and "grocer" each with one occurrence in the group. Note that the frequency of each keyword in the new pseudo-document is the sum of frequencies over all objects contained in the group. Consider a Boolean 1NN query to

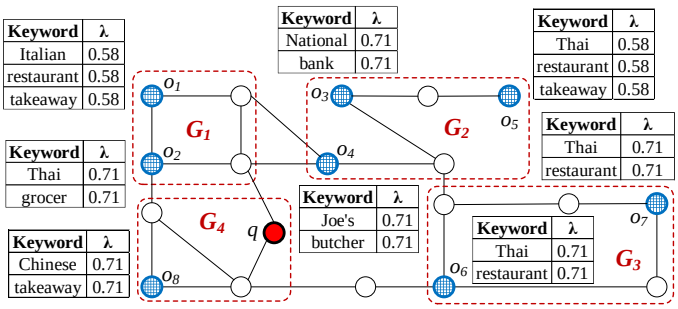


Fig. 1: Shaded vertices are objects and q is the query vertex.

find the closest object with keywords "Thai" *and* "restaurant". The group G_4 can be pruned as its pseudo-document does not contain "restaurant". The other three groups may contain the result. The algorithm computes minimum network distances to each group (e.g., the network distance from q to the closest border vertex in the group). These groups are inserted into a priority queue so that they can be accessed in ascending order of their minimum network distances from q (e.g., G_1, G_2, and then G_3). When G_1 is accessed, the algorithm prunes both o_1 and o_2 because neither object contains both query keywords. The algorithm then accesses G_2 and prunes the objects o_3 and o_4. But object o_5 contains both query keywords, so the algorithm computes its network distance from q. The algorithm can terminate if the minimum network distance of the next entry in the queue is greater than the network distance from q to o_5. However, since the minimum network distance of G_3 is smaller, the algorithm accesses G_3 and computes the network distances from q to o_6 and o_7 and determines o_6 to be the closest object satisfying the keyword criteria. So o_6 is returned as the result since the queue is empty.

In the above, costly minimum network distances needed to be computed to groups even when 1) a group does not contain any objects satisfying the keyword criteria (e.g., G_1) because the aggregated group appeared to contain such an object; or 2) the relevant object in the group is actually quite far from the query and is not a result (e.g., o_5 in G_2) because it appeared to be close as the query was close to the aggregated group. Furthermore, when a group is accessed, the algorithm needs to compute network distances from q to all objects satisfying the criteria in the group even if they are not results (e.g., o_7 in G_3). Similar issues are faced in top-k queries because network distances must be computed to groups (and objects within) that have low textual similarity (e.g., G_1 and its objects) or large network distance (e.g., o_5 and o_7). Moreover, while we use a simple example here for easier exposition, keyword aggregation is hierarchical and that exacerbates these problems. It is important to note that these problems arise from the hierarchical aggregation of objects and their constituent keywords. They cannot be solved in straight-forwards ways due to the permanent loss of discriminating information that results from aggregation. We confirm this difficulty by attempting to improve G-tree in Section 7.4. Other techniques [2], [3] use similar ideas and face similar problems, which we elaborate in Section 8.

1.2 Contributions

We present the **K**eyword **Sep**arated **In**dexing (K-SPIN) framework. K-SPIN employs the idea of creating a separate index for each keyword. However, as we detail next, doing this without incurring prohibitive pre-processing cost is challenging. Here we describe how K-SPIN overcomes the problems in our motivating example and our solutions for reducing pre-processing costs.

Technique	Index Size (in GB)	Queries/second	
		Top-k	BkNN
K-SPIN [Our Method] + CH [10]	0.6 + 0.6	865	1021
K-SPIN [Our Method] + PHL [11]	0.6 + 15.8	3942	9869
Spatial Keyword G-tree [4]	2.7	266	178
ROAD [12]	4.5	83	✗
FS-FBS [2]	Dataset too large to build index		

TABLE 1: Comparison of index size and throughput (# of queries processed per second) on US road network dataset

Efficient Querying: Separate keyword indexes allow us to obtain an *on-demand inverted heap* for each keyword filled with candidate objects specifically relevant to that keyword. Each candidate object in the heap is ranked by lower-bound network distance between it and the query. These heaps can be used to avoid false positives and reduce unnecessary network distance computations. We explain our method using the running example in Figure 1.

For the Boolean 1NN query to find the POI containing "Thai" *and* "restaurant", our algorithm creates an on-demand inverted heap for a single keyword. We obtain a heap for the least frequent keyword as it contains fewer objects. Using the heap for "Thai" (e.g., o_2, o_6, o_5, and then o_7), the first object o_2 is pruned because it does not satisfy the keyword criteria. When o_6 is accessed, its exact network distance is computed as it contains both query keywords. If the network distance of o_6 is smaller than the lower-bound distance of the next object (i.e., o_5), the algorithm terminates reporting o_6 as the result. The use of a cheap lower-bound heuristic avoids or delays computing expensive network distances to candidate objects (e.g., we avoid it for o_2, o_5, and o_7). Notably, this approach avoids generating false positive groups (e.g., G_1 in keyword aggregated indexing). In Section 5 we describe how heaps need only be populated partially and maintained in an iterative lazy manner, thus avoiding computing lower-bounds to all objects that contain the keyword. Other spatial keyword queries also benefit from inverted heaps. For example, we propose the idea of a *pseudo lower-bound* to retrieve more relevant candidates for top-k queries in Section 4.2.

This translates into significantly better query throughput (# of queries processed per second) for K-SPIN based techniques in practice, as summarized in Table 1. FS-FBS cannot be constructed on this dataset due to prohibitive pre-processing cost; and on smaller datasets FS-FBS performs worse than our method K-SPIN (Section 7). We even show that K-SPIN is able to use G-tree's road network index more efficiently than G-tree's own query algorithm, confirming the reduction in false positives (Section 7.4).

Light-Weight Separated Index: Creating a separate index for each keyword involves processing the objects and road network repeatedly. At first glance, doing this on road networks appears untenable. But every cloud has a silver lining. We make several smart, yet simple, observations (Section 6), which K-SPIN exploits to make the pre-processing more than viable, even light-weight. For example, we observe that the number of objects associated with a keyword is predictably small for most keywords in datasets that follow Zipf's law. Given the nature of K-SPIN, this observation can be leveraged to significantly reduce construction time with theoretical and experimental justification. We also introduce a novel data structure, the ρ-Approximate Voronoi Diagram, to reduce the index size by over an order of magnitude. K-SPIN is applicable on even continental scale datasets, occupying less than 600MB and built in under 2 hours for the entire US road network dataset. These come at a small and theoretically bounded penalty in query performance and we still return *exact* query results.

Flexibility: As the light-weight keyword indexes are decoupled from the network distance index, K-SPIN can be combined with any network distance technique. This enables significant performance gains, e.g., the K-SPIN variant using Pruned Highway Labeling (PHL) [11] in Table 1. Even the variant with the smallest memory footprint using Contraction Hierarchies (CH) [10] is much faster than the state-of-the-art in Table 1. Moreover, K-SPIN can be integrated into any system already processing road network queries and any future improved network distance technique can be plugged into the framework. This versatility of K-SPIN is unique among spatio-textual techniques.

2 PRELIMINARIES

Road Network: We represent a road network as a connected graph $G = (V, E)$. V is the set of vertices and E is the set of edges (i.e., road segments) connecting them. Similar to almost all previous studies (e.g., [2], [4]), we consider undirected edges and query locations and POIs occurring on vertices to make exposition simpler. As queries are graph operations, this does not change the asymptotic behavior. K-SPIN can easily be extended for other cases, e.g., POIs on edges would still be generated as candidates in on-demand inverted heaps.

An edge $(u, v) \in E$ connects two adjacent vertices with weight $w(u, v) \in \mathbb{R}_{>0}$, representing a metric such as travel time between u and v. The shortest path $P(x, y)$ with network distance $d(x, y)$ represents the minimum sum of edge weights connecting the vertices x and y.

Objects and Textual Information: The road network is also associated with a set of object vertices $O \subseteq V$ (i.e., POIs). Each object $o \in O$ contains a set of keywords known as the document, $doc(o)$, of object o. Each keyword $t \in doc(o)$ is drawn from a corpus of keywords W. For simplicity, we shall refer to $t \in doc(o)$ as $t \in o$ when the context is clear. We note that a keyword t may occur multiple times in $doc(o)$, the number of occurrences is denoted as its frequency $f_{t,o}$. Finally, the inverted list $inv(t)$ for keyword t is the set of objects whose document contains t. Next, we formally state our problem definitions.

Boolean kNN Queries: A Boolean k Nearest Neighbor (BkNN) query takes the form (q, k, ψ, op), where q is the query vertex, k is the number of results, ψ is a set of query keywords, and op specifies a logical operand ($\wedge$ or $\vee$) [2]. The result of this query is the k nearest objects by their network distance to q, which satisfy the criteria. In the conjunctive case ($\wedge$), the result objects must contain all query keywords; and in the disjunctive case ($\vee$), they contain at least one keyword from ψ. We remark that our proposed framework can be used to handle a combination of $\wedge$ and $\vee$ operators, e.g., find k closest POIs that contain "Thai" and ("takeaway" or "restaurant").

Top-k Spatial Keyword Queries: A top-k query is of the form (q, k, ψ), where q is the query vertex, k is the number of results, and ψ is a set of query keywords. The result is the set of k objects with the smallest scores. The score of each object is computed by combining its network distance from q and its textual relevance. We employ *weighted distance* [3], [13] to compute the spatio-textual score for object o, as below.

$$ST(q, o) = \frac{d(q, o)}{TR(\psi, o)} \tag{1}$$

Here, $d(q, o)$ is the network distance from q to o and $TR(\psi, o)$ is the textual relevance. We adopt *cosine similarity* [14] for computing $TR(\psi, o)$.

$$TR(\psi, o) = \frac{\sum_{t \in \psi}(w_{t,o} \cdot w_{t,\psi})}{\sqrt{\sum_{t \in o}(w_{t,o})^2 \cdot \sum_{t \in \psi}(w_{t,\psi})^2}} \tag{2}$$

In the above equation, $w_{t,o} = 1 + ln(f_{t,o})$ with $f_{t,o}$ being the frequency of keyword t in the document of o. Also, $w_{t,\psi} = ln(1 + \frac{|O|}{|inv(t)|})$, where $|O|$ is the total number of objects and $|inv(t)|$ is the size of the inverted list of t (i.e., the number of objects that contain t in their documents). While we do not dwell on the specifics of the textual relevance computation, $w_{t,o}$ represents a measure of the *term frequency* (TF), and $w_{t,\psi}$ represents the *inverse document frequency* (IDF).

As derived in past work [14], Equation 2 can be re-written in terms of *impacts*, or $\lambda_{t,x} = \frac{w_{t,x}}{\sqrt{\sum_{t \in x}(w_{t,x})^2}}$, as below.

$$TR(\psi, o) = \sum_{t \in \psi} [\lambda_{t,\psi} \cdot \lambda_{t,o}] \tag{3}$$

It is important to note that the object impact values $\lambda_{t,o}$ do not depend on the query and can be pre-computed offline. We emphasize that our indexing algorithms can support other textual relevance methods, such as language models, BM25, and other TF$\times$IDF formulations, e.g., in [14]. Similarly, our techniques are orthogonal to the scoring method and can be applied when *weighted sum* [8] is used to combine $d(q, o)$ and $TR(\psi, o)$ in Equation 1 instead of weighted distance, which we use as the example in our experiments.

3 K-SPIN: AN OVERVIEW

The modules that compose the K-SPIN framework are shown in Figure 2. Here we briefly describe each module and how they interact before delving deeper into the design of specific modules in subsequent sections.

1. Lower Bounding Module. This module computes a lower-bound network distance between any two vertices using selected heuristics. For example, a lower-bound can be obtained using landmarks as in the ALT [15] index. ALT pre-computes network distances between some chosen landmark vertices and all vertices in the graph, then uses the triangular inequality to obtain a lower-bound network distance between any two vertices. In fact, multiple heuristics can be considered to allow the module to return the tightest lower-bound network distance overall. Depending on the application and indexes available, the module may use more or fewer lower-bound heuristics. We combine K-SPIN with ALT as it provides effective lower-bounds on road networks [16].

2. Network Distance Module. This module is employed to compute the exact network distance between any two given vertices. As stated in Section 1, this module can use *any* existing technique to compute the network distance. The system administrator may choose a technique based on its efficiency and/or index size or may simply choose the techniques already being used by the system to answer other queries. In Section 7, we show the effect of choosing three different network distance techniques namely Contraction Hierarchies [10], G-tree [17], and Pruned Highway Labeling [11]. This module is the bottleneck as network distance computations are the most expensive operation performed for an object.

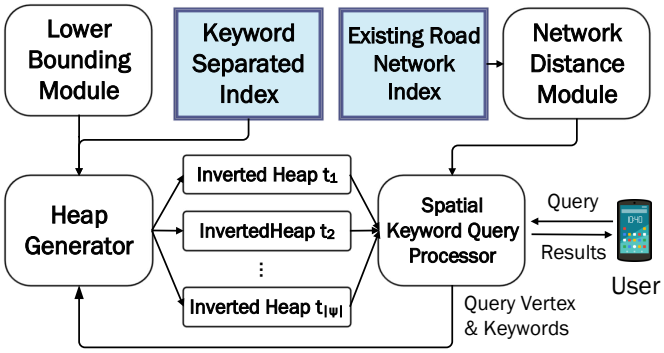


Fig. 2: Keyword Separated Indexing (K-SPIN) Framework

3. Heap Generator. The Heap Generator is responsible for creating and maintaining the *on-demand inverted heaps*. An on-demand inverted heap for a particular keyword t satisfies the following property at any point in time (i.e., when the heap is first created and whenever a heap element is extracted).

Property 1. Given the current top object o in inverted heap $\mathcal{H}$ for keyword t and its lower-bound distance $LB(q, o)$ from query vertex q; any object o_t containing t, not yet extracted from $\mathcal{H}$, has network distance $d(q, o_t) \geq LB(q, o)$.

Property 1 allows our query algorithms to access objects associated with a particular keyword t in order of their lower-bound network distances from q. To efficiently create and maintain an inverted heap, the Heap Generator utilizes a *Keyword Separated Index* that indexes $inv(t)$ for each keyword t in corpus W where $inv(t)$ is the set of all objects associated with t. E.g., for keyword "Thai" in Figure 1, inv("Thai") consists of o_2, o_5, o_6, and o_7. The inverted heap $\mathcal{H}_{Thai}$ allows access to these objects in ascending order of their lower-bounds, e.g., $(o_2, 1), (o_6, 2), (o_5, 4)$, and then $(o_7, 5)$. Property 1 allows the heap to be populated lazily, i.e., objects are added incrementally such that the property is met. For example, heap $\mathcal{H}_{Thai}$ may initially contain only $(o_2, 1)$ and $(o_7, 5)$ to satisfy Property 1. When $(o_2, 1)$ is extracted, the object $(o_6, 2)$ may be inserted in the heap to ensure it satisfies Property 1. Our Heap Generator algorithm and Keyword Separated Index data structure are presented in Sections 5 and 6, respectively.

4. Query Processor. The Query Processor contains algorithms to answer various spatial keyword queries. Algorithms use on-demand inverted heaps to retrieve relevant candidate objects. The challenge lies in deciding which heap to use and how to filter poor candidates using an effective lower-bound score. Hence the efficiency of the Query Processor is critical in avoiding the false positive problems of existing methods described in Section 1. The Query Processor uses the *Network Distance Module* to compute the network distances between the query vertex and the filtered candidate objects. Our query algorithms are detailed in Section 4.

4 The Query Processor Module

We first describe the algorithm for Boolean kNN queries in Section 4.1, demonstrating how inverted heaps are used. Section 4.2 details our top-k algorithm where we introduce the idea of a pseudo lower-bound utilizing a subtle insight to retrieve more relevant candidates and thereby terminating quicker.

4.1 Boolean kNN Query Processing

Boolean kNN (BkNN) queries retrieve the k nearest objects to q whose associated keywords satisfy some criteria with the set of query keywords ψ. In disjunctive queries reported objects contain at least one keyword in ψ and in conjunctive queries reported objects contain all keywords in ψ.

Algorithm 1 Disjunctive BkNN Query Processor

```
1:  function GetDisjunctiveBkNNs(k, q, ψ)
2:    Create on-demand inverted heap H_i for each keyword t_i ∈ ψ
3:    Initialize minimum priority queue PQ and set D_k ← ∞
4:    Insert minimum lower-bound distance for each H_i into PQ
5:    while !Empty(PQ) and MinKey(PQ) < D_k do
6:      H_s ← Extract-Min(PQ)
7:      LB(q, c) ← MinKey(H_s) , c ← Extract-Min(H_s)
8:      Call LazyReheap(H_s) to ensure Prop. 1 (see Section 6)
9:      Insert(PQ,[t_s,MinKey(H_s)])
10:     if c not already evaluated then
11:       Compute network distance d(q, c)
12:       if d(q, c) < D_k then
13:         Insert(L, [c, d(q, c)]) and update L and D_k if needed
14:   return L
```

4.1.1 Disjunctive Boolean kNN Queries

Algorithm 1 begins by initializing an on-demand inverted heap $\mathcal{H}_i$ for each keyword t_i (line 2). Recall that the Heap Generator ensures each heap $\mathcal{H}_i$ satisfies Property 1. Thus we access objects from each heap in order of minimum lower-bound distance from q. Priority queue $\mathcal{PQ}$ is used to choose the heap with the *smallest* minimum lower-bound distance. $\mathcal{PQ}$ is first initialized by inserting the lower-bound distance of the top object in each heap $\mathcal{H}_i$ (lines 3 and 4). The top element in $\mathcal{PQ}$ is extracted (line 6) to identify the heap $\mathcal{H}_s$ whose top object has the smallest lower-bound distance. Candidate object c is then extracted from $\mathcal{H}_s$ (line 7) and LazyReheap is called (line 8) to ensure $\mathcal{H}_s$ continues to satisfy Property 1 (detailed in Section 6). c is ignored if it has been extracted from another heap $\mathcal{H}_i$, otherwise network distance $d(q, c)$ is computed (line 11). The set of best candidates L seen so far is updated if c improves on it and D_k is also updated if needed (line 13). D_k corresponds to the distance of the kth closest object in L that satisfies the keyword criteria. An element for $\mathcal{H}_s$ is re-inserted into $\mathcal{PQ}$ with its new minimum lower-bound after c is extracted (line 9) to ensure $\mathcal{PQ}$ always chooses the heap whose top element has the smallest lower-bound distance. The algorithm terminates when $\mathcal{PQ}$ is empty or the next candidate object has a lower-bound distance greater than or equal to D_k (line 5).

Example 1. Consider a disjunctive B1NN query from vertex q in Figure 1 with query keywords "Thai" and "restaurant". On-demand inverted heaps are generated for each keyword, e.g., $\mathcal{H}_{Thai}$=$\{(o_2, 1), (o_6, 2)\}$ and $\mathcal{H}_{rest.}$=$\{(o_1, 2), (o_6, 2)\}$. Each heap's minimum lower-bound is inserted into the priority queue, so $\mathcal{PQ}$=$\{(Thai, 1), (rest., 2)\}$. The top element $Thai$ from $\mathcal{PQ}$ is extracted, identifying $\mathcal{H}_{Thai}$. Now the top candidate object o_2 is extracted from inverted heap $\mathcal{H}_{Thai}$. Naturally, o_2 satisfies the disjunctive criteria, so its network distance is computed, added to the result set L and D_k is set to $d(q, o_2)$=2. LazyReheap is called to ensure $\mathcal{H}_{Thai}$ satisfies Property 1, e.g., $\mathcal{H}_{Thai}$=$\{(o_6, 2), (o_7, 4)\}$. $Thai$ is reinserted into $\mathcal{PQ}$ with the minimum lower-bound in $\mathcal{H}_{Thai}$, so $\mathcal{PQ}$=$\{(Thai, 2), (rest., 2)\}$. L cannot be improved as MINKEY($\mathcal{PQ}$) is equal to D_k and Algorithm 1 terminates.

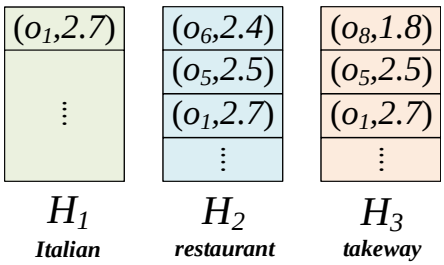


Fig. 3: Computing Pseudo Lower-Bounds on Inverted Heaps

4.1.2 Conjunctive Boolean kNN Queries

A similar algorithm can be created for conjunctive BkNN queries. We discuss it briefly due to space constraints (it is included in our experiments). The basic idea is to only use the inverted heap for the least frequent keyword as it has the fewest objects. As candidate objects are retrieved, those without all keywords are filtered, avoiding costly network distance computations.

4.2 Top-k Query Processing

We propose a novel top-k query algorithm to retrieve the k objects with the best spatio-textual scores by Eq. (1). Our algorithm computes the top-k objects by utilizing, for each inverted heap, a *pseudo lower-bound score* on only some of the unseen objects in the heap. The algorithm still computes correct results even though the pseudo lower-bound score is not a valid lower-bound score for all unseen objects in the heap. Next, we first describe how to compute a valid lower-bound score on all unseen objects.

Valid Lower-Bound Score on All Unseen Objects: Consider a top-k query for three keywords "Italian", "restaurant", and "takeaway". Let $TR_{max}(\psi, P)$ be the maximum possible textual relevance for query keywords ψ with any object in set P. For simplicity, assume that textual similarity $TR(\psi, o)$ is the number of query keywords present in the object o, so $TR_{max}(\psi, P) = 3$ in this example. Figure 3 shows the three inverted heaps $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$ created for this query with objects from our running example in Figure 1. Since we do not know the textual similarity of unseen objects in any heap $\mathcal{H}_i$, a lower-bound score for all unseen objects in $\mathcal{H}_i$ can be computed using the maximum textual similarity and minimum lower-bound distance in the heap as $ST_{all}(\psi, \mathcal{H}_i) = \frac{\text{MINKEY}(\mathcal{H}_i)}{TR_{max}(\psi,P)}$. For example, the best possible score for any unseen object in $\mathcal{H}_1$ is $\frac{LB(q,o_1)}{3} = \frac{2.7}{3} = 0.9$, $\mathcal{H}_2$ is $\frac{2.4}{3} = 0.8$, and $\mathcal{H}_3$ is $\frac{1.8}{3} = 0.6$. But, as we explain next, it is possible to obtain a pseudo lower-bound score tighter than this without losing any top-k results.

A Key Insight: Heap $\mathcal{H}_3$ in Figure 3 has the smallest lower-bound distance and its top element is object o_8. Since the lower-bound distance $LB(q, o_8) = 1.8$ is smaller than the lower-bound distance of $\mathcal{H}_1$ (i.e., $LB(q, o_1) = 2.7$), this implies that either o_8 has been extracted from $\mathcal{H}_1$ or o_8 does not contain the keyword "Italian". For the same reason, o_8 has either been extracted from $\mathcal{H}_2$ or does not contain the keyword "restaurant". In other words, either o_8 has already been processed by the algorithm (i.e., extracted from another heap) or o_8 only contains the keyword "takeaway". Similarly for heap $\mathcal{H}_2$, top element o_6 has either been extracted from $\mathcal{H}_1$ or at best contains only the keywords "restaurant" and "takeaway" (o_6 may contain "takeaway" because o_6 may still be in $\mathcal{H}_3$ as it has a smaller top lower-bound distance).

Pseudo Lower-Bound Score: Using the insight above, we compute a *pseudo lower-bound score* that is a lower-bound score on a subset of the objects in the inverted heap and hence is tighter than the valid lower-bound score. Let $\text{MINKEY}(\mathcal{H}_j)$ be the minimum lower-bound network distance for an element in heap $\mathcal{H}_j$. If $\mathcal{H}_j$ has become empty, $\text{MINKEY}(\mathcal{H}_j)$ is assumed to be infinite. The pseudo lower-bound score for a heap $\mathcal{H}_i$ is computed by assuming that every unseen object in $\mathcal{H}_i$ contains a keyword t_j only if $\text{MINKEY}(\mathcal{H}_i) \geq \text{MINKEY}(\mathcal{H}_j)$ where t_j is the query keyword associated with heap $\mathcal{H}_j$. We next describe the algorithm to compute pseudo lower-bound scores.

Algorithm 2 Compute Pseudo Lower-Bound Score for $\mathcal{H}_i$

```
1: function PSEUDOLB(ψ, H_i)
2:     TR_p(ψ, H_i) ← 0
3:     for each keyword t_j ∈ ψ do
4:         if MINKEY(H_i) ≥ MINKEY(H_j) then
5:             TR_p(ψ, H_i) ← TR_p(ψ, H_i) + λ_{t_j,ψ} × λ_{t_j,max}
6:     return ST_pLB(H_i) ← MINKEY(H_i) / TR_p(ψ, H_i)
```

Algorithm 2 computes the pseudo lower-bound score for inverted heap $\mathcal{H}_i$ denoted by $ST_{pLB}(\mathcal{H}_i)$. First, it computes a pseudo textual relevance $TR_p(\psi, \mathcal{H}_i)$ following Eq. 3 by considering only the keywords that satisfy the condition described above (lines 4-5). Note that we use the real maximum impact $\lambda_{t_j,max}$ of t_j in any object, which can be cheaply computed offline for all keywords. The pseudo lower-bound score is computed using the pseudo textual relevance $TR_p(\psi, \mathcal{H}_i)$ and then returned (line 6).

Example 2. Consider again the example in Figure 3. The pseudo lower-bound score of $\mathcal{H}_2$ is computed assuming that all unseen objects in $\mathcal{H}_2$ can only include two keywords "restaurant" and "takeaway", i.e., $TR_p(\psi, \mathcal{H}_2) = 2$ and $ST_{pLB}(\mathcal{H}_2) = \frac{2.4}{2} = 1.2$. Similarly, $\mathcal{H}_3$ includes only the keyword "takeaway" and $ST_{pLB}(\mathcal{H}_3) = \frac{1.8}{1} = 1.8$. $\mathcal{H}_1$ includes all three keywords and $ST_{pLB}(\mathcal{H}_1) = \frac{2.7}{3} = 0.9$. Note that pseudo lower-bound scores are not valid lower-bound scores, e.g., the spatio-textual score of o_1 in $\mathcal{H}_2$ is $\frac{d(q,o_1)}{TR(\psi,o_1)} = \frac{3}{3} = 1$ which is smaller than the pseudo lower-bound $ST_{pLB}(\mathcal{H}_2) = 1.2$.

Next, we show how the Query Processor can use pseudo lower-bounds to answer top-k queries instead of valid lower-bounds. We then prove that it still computes correct results and elaborate on why the pseudo lower-bound score is useful.

Query Processor: The top-k algorithm (Algorithm 3) is quite similar to the algorithm for computing disjunctive BkNN queries. The main difference is that pseudo lower-bound scores of heaps are used in $\mathcal{PQ}$ (see line 4) to access the heap with the best candidate object. If the extracted candidate object c has not already been processed, a lower-bound score is cheaply computed using its *actual* textual relevance and lower-bound network distance (line 10), i.e., $\frac{LB(q,c)}{TR(\psi,c)}$. If this lower-bound score is smaller than D_k, then its actual score is computed using its exact network distance $d(q, c)$ (lines 11 and 12). If its actual score is smaller than D_k, the result list L and D_k are updated accordingly (line 14). The algorithm terminates when $\mathcal{PQ}$ is empty or the top of $\mathcal{PQ}$, representing the smallest pseudo lower-bound score of any heap, is greater than or equal to D_k as L can no longer be improved.

Example 3. Consider a top-1 query for our running example in Figure 1 and Figure 3 with keywords "Italian", "restaurant", and "takeaway". If the heaps are accessed considering the actual lower-bounds, o_1 (which is the result) will be the last accessed object. However, our algorithm accesses the heaps

based on their pseudo lower-bound scores and $\mathcal{H}_1$ has smaller pseudo lower-bound scores than the other two heaps (as seen in Example 2). Thus $\mathcal{H}_1$ is accessed first. So Algorithm 3 extracts candidate o_1, computes its spatio-textual score $\frac{d(q,o_1)}{TR(\psi,o_1)} = 1$, re-inserts an element into $\mathcal{PQ}$ for H_1 with its new MINKEY (i.e., infinity as $\mathcal{H}_1$ is now empty). After updating the result set with o_1, the algorithm then terminates because the next best pseudo lower-bound score in $\mathcal{PQ}$ is $ST_{pLB}(\mathcal{H}_2) = 1.2$ which is higher than the score of the current top-1 object o_1.

Implementation Notes: While the same candidate may be extracted from multiple heaps (i.e., when associated with multiple query keywords), these can be ignored by using a hash-table or bit-array to track evaluated candidates. In any case, this only entails a small query overhead as the lower-bound computation is cheap and the heap only contains a small number of objects (due to being lazily populated) resulting in a small update cost. Also note that query impacts $\lambda_{t,\psi}$ need only be computed once for the query and $TR(\psi, c)$ need only be computed once for each candidate.

Benefits of Pseudo Lower-Bound Scores: We propose Lemma 1 to show that a pseudo lower-bound score is never worse than the valid lower-bound score for all unseen objects.

Lemma 1. For any heap $\mathcal{H}_i$, the pseudo lower-bound is always greater than or equal to the valid lower-bound for all unseen objects in $\mathcal{H}_i$, i.e., $ST_{pLB}(\psi, \mathcal{H}_i) \geq ST_{all}(\psi, \mathcal{H}_i)$.

Proof: Since $ST_{pLB}(\psi, \mathcal{H}_i) = \frac{\text{MINKEY}(\mathcal{H}_i)}{TR_p(\psi,\mathcal{H}_i)}$ and $ST_{all}(\psi, \mathcal{H}_i) = \frac{\text{MINKEY}(\mathcal{H}_i)}{TR_{max}(\psi,P)}$, it suffices to show that $TR_p(\psi, \mathcal{H}_i) \leq TR_{max}(\psi, P)$. The maximum possible textual relevance for any object in set P can be computed by $TR_{max}(\psi, P) = \sum_{t\in\psi} \lambda_{t,\psi} \times \lambda_{t,max}$ where $\lambda_{t,max}$ is maximum impact of keyword t in any object. By Algorithm 2, we have $TR_p(\psi, \mathcal{H}_i) = \sum_{t_j\in\psi} \lambda_{t_j,\psi} \times \lambda_{t_j,max}[\text{MINKEY}(\mathcal{H}_i) \geq \text{MINKEY}(\mathcal{H}_j)]$ where t_j is the keyword associated with heap $\mathcal{H}_j$. Clearly the maximum value of $TR_p(\psi, \mathcal{H}_i)$ is $TR_{max}(\psi, P)$, occurring when condition $[\text{MINKEY}(\mathcal{H}_i) \geq \text{MINKEY}(\mathcal{H}_j)]$ evaluates to true for all heaps $\mathcal{H}_j$. Thus $TR_p(\psi, \mathcal{H}_i) \leq TR_{max}(\psi, P)$, thereby completing the proof. □

From Lemma 1, it can be seen that the textual relevance used for a pseudo lower-bound depends on the condition $[\text{MINKEY}(H_i) \geq \text{MINKEY}(H_j)]$ over all j. This condition is likely to result in decreasing textual relevance for each subsequent heap in descending order of their MINKEY values. This entails increasing pseudo lower-bounds, which in turn allows Algorithm 3 to avoid accessing heaps and terminate sooner. Conversely, the pseudo lower-bound assigns higher textual relevance to larger MINKEY values. This allows Algorithm 3 to access more promising candidates, e.g., those that are far from q but contain all keywords and have high textual relevance. Furthermore, K-SPIN is likely to filter out any bad candidates using their actual textual relevance without computing expensive network distances. Pseudo lower-bounds can be applied to any textual model that computes similarity per query keyword, as many popular methods do, including language models, TF×IDF, and BM25.

Proof of Correctness: As stated earlier, pseudo lower-bound scores are not valid lower-bounds, e.g., $ST_{pLB}(\mathcal{H}_2) = 1.2$ is higher than the score of o_1 (1) which is also present in $\mathcal{H}_2$. While it may seem like this can lead to missing objects, the algorithm still produces correct results, e.g, because o_1 is also present in $\mathcal{H}_1$ and its score cannot be better than $ST_{pLB}(\mathcal{H}_1)$. We propose Lemma 2 to express this formally.

Algorithm 3 Top-k Query Processor

```
1: function GETTOPKOBJECTS(q, k, ψ)
2:   Create on-demand inverted heap H_i for each keyword t_i ∈ ψ
3:   Initialize minimum priority queue PQ and set D_k ← ∞
4:   Insert pseudo lower-bound score for each H_i into PQ
5:   while !EMPTY(PQ) and TOP(PQ) < D_k do
6:     n ← EXTRACT-MIN(PQ)
7:     LB(q, c) ← MINKEY(H_n) , c ← EXTRACT-MIN(H_n)
8:     LAZYREHEAP(H_n)
9:     INSERT(PQ,[n,PSEUDOLB(ψ, H_n)])
10:    if c not already processed or LB(q,c)/TR(ψ,c) ≤ D_k then
11:      Compute network distance d(q, c)
12:      ST(q, c) ← d(q,c)/TR(ψ,c)        ▷ Compute actual score
13:      if ST(q, c) < D_k then
14:        INSERT(L, (c, ST(q, c))), update L and D_k if needed
15:  return L
```

Lemma 2. When Algorithm 3 terminates, every object o that has not been seen has $ST(q, o) \geq D_k$.

Proof: The algorithm terminates when $\text{TOP}(\mathcal{PQ}) \geq D_k$. This implies that, for every heap $\mathcal{H}_i$, $ST_{pLB}(\mathcal{H}_i) \geq D_k$ when the algorithm terminates. Let $\mathcal{H}_{max}$ be the heap with the largest MINKEY. Next, we show that $ST(q, o) \geq ST_{pLB}(\mathcal{H}_{max})$ which implies that $ST(q, o) \geq D_k$ for every unseen object o.

Recall that $ST(q, o) = \frac{d(q,o)}{TR(\psi,o)}$ and $ST_{pLB}(\mathcal{H}_{max}) = \frac{\text{MINKEY}(\mathcal{H}_{max})}{TR_p(\psi,\mathcal{H}_{max})}$. Since $\mathcal{H}_{max}$ is the heap with the largest MINKEY, Algorithm 2 (lines 4-5) computes $TR_p(\psi, \mathcal{H}_{max})$ assuming it contains all query keywords. Therefore, $TR(\psi, o) \leq TR_p(\psi, \mathcal{H}_{max})$. Furthermore, since o has not been seen by Algorithm 3, $LB(q, o) \geq \text{MINKEY}(\mathcal{H}_{max})$ otherwise it would have been extracted from at least one heap $\mathcal{H}_i$. Thus, $d(q, o) \geq \text{MINKEY}(\mathcal{H}_{max})$. Hence, $ST(q, o) \geq ST_{pLB}(\mathcal{H}_{max})$.

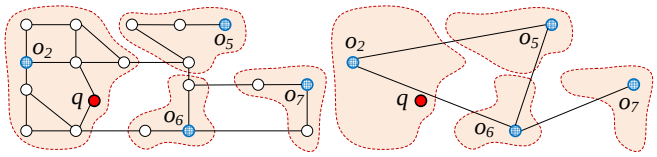
□

5 HEAP GENERATOR MODULE

A Heap Generator creates an on-demand inverted heap for query keyword t. This inverted heap satisfies Property 1, i.e., allows access to objects containing keyword t in ascending order of their lower-bound network distances from query location q. A simple approach to ensure Property 1 is to insert all objects from the inverted list of t (i.e., $inv(t)$) in the heap with their lower-bound distances. However this is not feasible as it would be required for *every* query. In this section we describe a Heap Generator based on the Network Voronoi Diagram (NVD) [18] that instead allows inverted heaps to be populated lazily. However NVDs possess high pre-processing costs, which we describe below before proposing a solution (with low pre-processing cost) in Section 6.

Network Voronoi Diagrams: Given a set of objects $inv(t)$ containing keyword t, an NVD is a disjoint partitioning of the road network vertices V for each object in $inv(t)$. A partition for object o_i is the Voronoi node set $Vns(o_i) \subseteq V$ which contains every vertex for which o_i is its closest object by network distance. After computing all Voronoi node sets, the NVD stores the nearest object o_i for every vertex in V.

Figure 4(a) shows the NVD for the set of objects containing keyword "Thai" from our running example. The shaded containers indicate the vertices belonging to each Voronoi node set. Note that



(a) NVD for "Thai" in Fig. 1 (b) Adjacency Graph for Fig. 4(a)

Fig. 4: Example Network Voronoi Diagram

the NVD does not depend on the query vertex q. Two Voronoi node sets $Vns(o_i)$ and $Vns(o_j)$ are considered *adjacent*, if there is an edge $(u, v) \in E$ connecting $u \in Vns(o_i)$ and $v \in Vns(o_j)$. For simplicity, we also say that o_i and o_j are adjacent. So, in Figure 4(a), o_2 and o_5 are adjacent as there is an graph edge connecting the shaded containers. Similarly, o_2 and o_6 are also adjacent.

NVD-Based kNN Algorithms [16], [18]: The 1NN of a query vertex can be found using an NVD as it stores the nearest object o_i for each vertex, e.g., o_2 is the 1NN of q in Figure 4(b) as q is in its Voronoi node set (shaded container). Kolahdouzan *et al.* [18] presented a useful property to find kNNs.

Property 2. k-th nearest object of q must be an object adjacent (in the NVD) to the first $k-1$ nearest objects of q.

For example, the 2nd NN of q must be among the objects adjacent to o_2 (i.e., o_5 and o_6). This is because the shortest path from q to the 2nd NN must leave $Vns(o_2)$ and enter one of the adjacent Voronoi node sets. Existing techniques [16], [18] exploit this property to incrementally answer kNN queries.

Heap Generation via NVD: Property 2 can also be used to create and lazily maintain an on-demand inverted heap for any keyword t. Specifically, a heap can be initialized by inserting 1NN of q obtained from the NVD. Then, whenever an object o is extracted, the adjacent objects of o in the NVD are inserted into the heap with their lower-bound network distances.

When an NVD is computed, we also create an *adjacency graph* representing the relationships between objects that are adjacent to each other. Each node in the adjacency graph is an object o_i and an edge between two nodes o_i and o_j is created if o_i and o_j are adjacent in the NVD. Figure 4(b) shows an adjacency graph for the NVD shown in Figure 4(a).

Algorithm 4 describes how to maintain an inverted heap $\mathcal{H}$. After the heap is initialized with the 1NN as described earlier, LAZYREHEAP is called whenever an object o_c is extracted from $\mathcal{H}$. Then the adjacent objects of o_c that were not previously inserted are now inserted in $\mathcal{H}$ with their lower-bound network distances using the NVD's adjacency graph.

Limitations: While NVDs allow efficient creation and maintenance of on-demand inverted heaps, it comes at the expense of higher pre-processing cost. This is exacerbated by building an NVD for each keyword t. An NVD takes $O(|V|log|V|)$ time and $O(|V|)$ space [19] and building one for each keyword multiplies them by $|W|$. For example, $|V|$ is 24 million and $|W|$ is $106,000$ for the US road network dataset and, even with existing optimizations, the resulting index takes 3-days to be build and occupies 90GB of memory! Updating NVDs when an object is added/deleted or changed also comes at a sizable cost. Next, we make several important observations and propose a space-efficient NVD with significantly reduced pre-processing and update cost.

Algorithm 4 Heap Maintenance Algorithm

```
1: function LazyReHeap(H, q, o_c)
2:   for each o_a adjacent to o_c in adjacency graph do
3:     if o_a has not been inserted into H then
4:       Compute lower-bound network distance LB(q, o_a)
5:       H.Insert(o_a, LB(q, o_a))
6:       Mark o_a as "inserted" into H
```

5.1 Query Processor Complexity

Based on this heap generator module, we may now derive expressions for query time. We perform our analysis for BkNN queries, but similar analysis can be performed for top-k queries. For a BkNN query, let us say the loop in Algorithm 1 runs for $\kappa \geq k$ iterations. The value of κ depends on the efficiency of the candidate generation heuristic. The inverted heap $\mathcal{H}$ contains at most $|O|$ objects, thus extracting from a binary heap implementation takes $O(\log |O|)$ time. For an NVD graph with maximum degree Δ, LAZYREHEAP computes a lower-bound for each adjacent object and inserts them into heap $\mathcal{H}$ at cost $O(\log |O|)$. Using the ALT index to compute a lower-bound takes $O(m)$ time where m is a small constant (typically 16) and in practice Δ is also a small constant both in our experiments and past studies [18]. Lastly, a single network distance computation is performed per iteration with time, denoted by $O(NDIST)$, depending on the technique used. This operation tends to dominate the iteration's cost, e.g., a Contraction Hierarchies query takes $O(\log^2 n \log^2 D)$ time [20]. So the total query time is $O(\kappa m \Delta \log |O| + \kappa NDIST)$ for a BkNN query. Top-k queries have an additional small constant time cost per iteration to compute textual relevance. The smallest possible value of κ is k for a perfect heuristic and in practice κ is a small constant multiple of k, at most $3k$ for BkNN and $5k$ for top-k queries over all settings in our experiments.

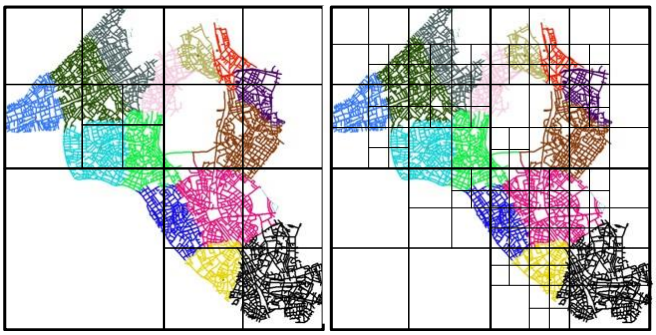
6 KEYWORD SEPARATED INDEX

Keyword separation has led to the higher pre-processing cost described above, but a remedy can also be found in keyword separation. Inspired by several simple but smart observations, we propose a novel space-efficient NVD. The resulting Keyword Separated Index is not only viable but also light-weight.

Observation 1: Most keywords have small inverted lists and this is consistent for any Zipfian dataset. Keywords in real-world datasets are known to follow Zipf's law [2]. Let frequency f_t be the size of a keyword t's inverted list and r_t be the rank of t by its frequency in corpus W. Zipf's law states $f_t \propto \frac{1}{{r_t}^\alpha}$ with $\alpha \approx 1$. In simple terms, classic Zipf's law suggests keyword t with rank r_t occurs $\frac{1}{r_t}$ as often as the most frequent keyword.

We can predict the frequency of any keyword using the theoretical basis of Zipf's law. For example, we can predict that 80% of keywords have a frequency of $\frac{f_{max}}{0.2|W|}$ or less, where f_{max} is the maximum frequency of a keyword and $|W|$ is the number of keywords. This predicted 80-th percentile frequency is less than or equal to 5 for all datasets listed in Table 2 and closely matches the real values. On reflection, this is not surprising as Zipifian distributions follow a harmonic progression, i.e., are "long-tailed".

K-SPIN can exploit this observation to avoid creating indexes for a vast majority of keywords while only paying a small penalty in query performance. If the number of objects in the inverted list $inv(t)$ of keyword t is at most a small constant ρ, we do not create an NVD at all. For queries involving such keywords,



(a) 4-approximate Quadtree (b) Region Quadtree (shown to depth 4)

Fig. 5: ρ-Approximate Network Voronoi Diagram

we simply need to initialize the inverted heap with all objects in $inv(t)$, which is at worst only ρ objects and in K-SPIN only costs a cheap lower-bound computation anyway. Using $\rho = 5$, indexes for over 80% of keywords are avoided, substantially reducing pre-processing cost. Moreover, given the long tail Zipfian distributions, such a ρ will scale slowly for increasing keyword dataset size.

Observation 2a: While the size of an NVD is $O(|V|)$, the adjacency graph takes $O(|inv(t)|)$ space where $|inv(t)| \leq |V|$ is the total number of objects containing keyword t. In general, the average degree in NVD adjacency graphs is a small constant, e.g., 6 as shown in [18] over several real-world road networks. Therefore, the adjacency graph's size is linear to the number of objects and is independent of $|V|$. In short, *we only need the small adjacency graph to maintain the heap and not the NVD which is the bottleneck for space usage*.

Observation 2b: K-SPIN does not actually require an NVD to provide the exact 1NN of q when initializing an inverted heap. The heap would still satisfy Property 1 if we initialize it with $\rho \geq 1$ candidate objects as long as the 1NN of q is among the ρ objects, as proven in Theorem 1.

Theorem 1. An inverted heap $\mathcal{H}$ initialized as above and maintained by Algorithm 4 satisfies Property 1. Specifically, let o_c be the current top object in $\mathcal{H}$ with lower-bound network distance $LB(q, o_c)$. Every object o_x that is not yet extracted from $\mathcal{H}$ has network distance $d(q, o_x) \geq LB(q, o_c)$.

Proof: We prove Theorem 1 for each possible case:

At Initialization: Let o_1 be the 1NN of q. At initialization, the heap contains up to ρ objects including o_1. Since o_1 is the 1NN, $d(q, o_1) \leq d(q, o_x)$. And since o_1 is in the heap, it is obvious that $LB(q, o_c) \leq LB(q, o_1) \leq d(q, o_1) \leq d(q, o_x)$ (note that o_1 and o_c could be the same object).

General Case: If o_x is in the heap then clearly we have $LB(q, o_c) \leq LB(q, o_x) \leq d(q, o_x)$. If o_x is not in the heap, this means o_x is not adjacent to any object that has been extracted from $\mathcal{H}$ (or it would have been inserted it into the heap by Algorithm 4). Therefore, as observed in [18], there exists at least one object o_y in the heap such that $d(q, o_y) \leq d(q, o_x)$. Since o_c is the top object in the heap we must have $LB(q, o_c) \leq LB(q, o_y)$, which implies $LB(q, o_c) \leq LB(q, o_y) \leq d(q, o_y) \leq d(q, o_x)$. □

Observation 3: Separated indexing means that building NVDs are independent operations. As an added benefit of the K-SPIN framework, NVD construction can be easily parallelized on all available cores to further reduce the construction time.

6.1 ρ-Approximate Network Voronoi Diagram

Observations 2a and 2b suggest that an exact NVD is not necessary to initialize or maintain inverted heaps. We propose the ρ-Approximate NVD, defined below, to take advantage of these observations while significantly reducing index size. Furthermore, due to the nature of K-SPIN, it still returns *exact* query results.

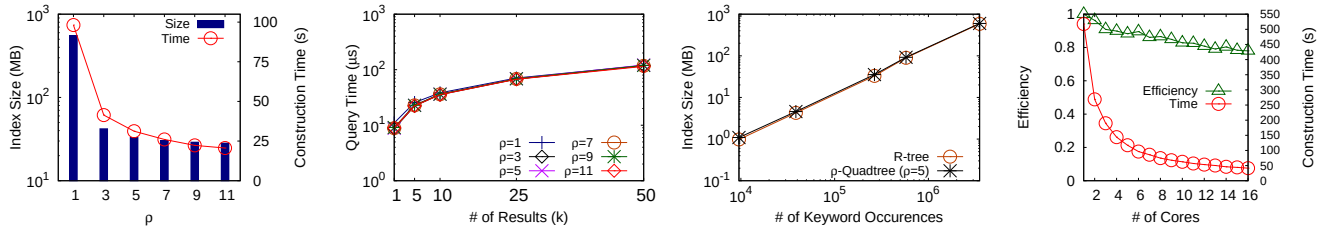
Definition 1. A ρ-Approximate Network Voronoi Diagram allows retrieving, for *every* vertex $v \in V$, up to ρ objects such that one of these ρ objects is the 1NN of v.

Constructing ρ-Approximate NVDs: We first compute an exact NVD in $O(|V| \log |V|)$ time if there more than ρ objects. We then store a ρ-Approximate NVD in a quadtree as follows. The root node of the quadtree is a minimum bounding box of all vertices in the road network. Each node is recursively divided into four children until all the vertices contained in the node belong to at most ρ different Voronoi node sets. To simplify the explanation, assume that each object $o \in O$ has a unique color and an NVD is represented by assigning each vertex $v \in V$ the same color as the color of its nearest object (see Figure 5). The ρ-Approximate quadtree continues dividing nodes into four children until the node contains at most ρ different colors. Figure 5(a) shows a 4-Approximate NVD indexed using a quadtree. After each iteration only the ρ-Approximate NVD is kept (the exact NVD is not kept). If there are fewer than ρ objects, the exact NVD does not need to be computed at all, which is quite beneficial as per Observation 1.

The ρ-Approximate NVD indexed using a quadtree significantly reduces space usage compared to an exact NVD indexed using standard techniques such as a region quadtree. By relaxing the need to distinguish the boundaries of different Voronoi node sets, the ρ-Approximate NVD is able to reduce the height of the required quadtree as shown in Figure 5(a). On the other hand, an exact NVD's region quadtree continues dividing nodes into four children until the node contains exactly one color, i.e., $\rho = 1$. This results in a deeper tree and hence significantly higher space usage. For example, in Figure 5(b) the exact NVD's region quadtree is shown only up to a depth of 4 and there are still quite a few nodes that contain more than 1 color. Voronoi node sets exhibit *spatial coherence* [21], forming largely contiguous regions. This property combined with Observation 2a, the number of adjacent Voronoi node sets being a small constant, suggest ρ-Approximate NVDs will be quite effective even for small values of ρ.

Experimental Index Size and Time: Figure 6(a) shows the effect of ρ from 1 to 11 on pre-processing of the Florida road network with 1 million vertices. Observation 2a+b result in an index that is 18 times smaller for $\rho = 5$ than exact NVDs indexed by region quadtrees (i.e., $\rho = 1$) as shown in the bar plots (refer to the left-hand y-scale). The effect of Observation 1 is seen in the index time line plot (refer to the right-hand y-scale), with a substantial reduction in construction time with increasing ρ. Florida is used as exact indexes (ρ=1) cannot be constructed on larger datasets.

Heap Initialization and Query Penalty Guarantee: The penalty paid for the approximation is during the heap initialization, when a point location query is issued on a ρ-Approximate NVD quadtree to find the cell containing q. Since the cell contains at most ρ colors, the Heap Generator computes the lower-bound network distances to at most ρ objects (the 1NN is among them) and inserts these in the heap. In the worst-case, when lower-bounds of the $\rho - 1$ objects are smaller than the lower-bound of the 1NN, the algorithm needs to compute network distances to these $\rho - 1$



(a) ρ vs. Pre-Processing (b) ρ vs. Top-k Query Time (c) R-trees vs. ρ-Quadtrees (d) Effect of Parallelization

Fig. 6: ρ-Approximate NVD Performance on Florida Road Network (# of terms=2, k=10)

objects. Thus, the worst-case penalty is $\rho - 1$ network distance computations. However in practice, these (at most) $\rho - 1$ objects are very likely to be adjacent objects to 1NN of q and thus would normally be evaluated as candidates anyway. This is verified in Figure 6(b) as query time does not vary for different ρ.

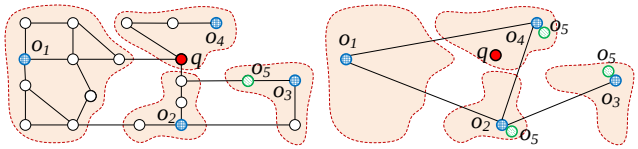
Space Complexity Theory vs. Practice: Approximate NVDs can alternatively be stored in R-trees. In this case, the leaf nodes of the R-tree contain the Minimum-Bounding Rectangle (MBRs) covering each Voronoi node set. However, R-trees cannot provide the ρ guarantee on the number of 1NN candidates as more than ρ MBRs may overlap and contain the query vertex q. On the other hand, R-trees can provide a worst-case space complexity. If $|inv(t)|$ is the number of objects indexed by the NVD for keyword t then there will be $|inv(t)|$ such MBRs, so the total space complexity for all keywords is $O(\sum_{t \in W} |inv(t)|)$. In other words, the space is linear in the total number of keyword occurrences, which is the space cost of the input keyword dataset.

Figure 6(c) shows the comparison of index size for a number of real-world road network datasets (Table 2), with the number of keyword occurrences increasing from left to right. As expected, the index size of Approximate NVDs stored in R-trees increases linearly with the number of keyword occurrences. Remarkably, storing in quadtrees also displays comparable and linearly increasing index size. While it remains to be seen whether quadtrees also theoretically take space linear in the number of keyword occurrences (the input), we see that this is true in practice on real datasets. So R-trees provide a worst-case guarantee on index size, while ρ-Approximate NVDs stored in quadtrees provide a guarantee on the number of 1NN candidates. Given the candidate guarantee, slightly faster construction and flexibility offered by ρ, we choose quadtrees in our experiments and represent them as Morton lists [22] which display better locality of reference.

Parallelized NVD Construction: Figure 6(d) exhibits significant speed-up using multi-core processing, with NVD construction time reduced by a factor of 12.5 with 16-cores. Efficiency ($\frac{T_1}{p * T_p}$ where T_p is the time for p cores) barely drops below 80%, suggesting serial parts of NVD construction are not significant and corroborating Observation 3. We parallelize NVD construction over all available cores in subsequent experiments.

6.2 Handling Updates

Our ρ-Approximate NVD index (called APX-NVD hereafter) can handle various types of object and keyword updates. Both insertion and deletion of either objects or keywords are ultimately handled in the same way, i.e., by adding/deleting objects to/from the APX-NVD of the affected keyword(s). For example, to incorporate a new object o with keyword set ψ, o is added to the NVD of each keyword $t \in \psi$. Similarly, adding/deleting a keyword t to/from an existing object o involves adding/deleting o to/from



(a) o_2, o_3, and o_4 are affected (b) Adjacency Graph after insertion

Fig. 7: Updating APX-NVD after inserting o_5

the NVD for t. In this section we present techniques to support these basic operations efficiently using *lazy updates*. Generally adding/deleting an object involves full or part re-computation of the NVD, which is a relatively expensive operation, e.g., requiring up to 1 second per NVD on the Florida dataset. We delay this re-computation by allowing a certain threshold of lazy updates to the APX-NVD while still supporting exact querying *and* amortizing the re-computation cost over multiple updates.

Object Deletion: Deleting object o from an APX-NVD is handled simply by marking o as deleted. If an extracted object is marked as deleted, the Heap Generator does not return it to the Query Processor. Its adjacent objects are still added to the heap as usual.

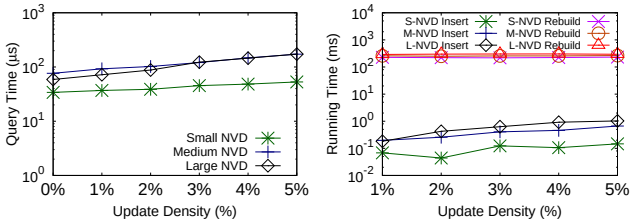
Object Insertion: Inserting an object o is more complicated and requires knowing the objects that might be affected by inserting o. We define *affected set* $A(o)$ as the objects whose Voronoi node sets may change when o is added to the NVD.

A previous study [18] reported that the affected set of o consists of the 1NN and its adjacent objects. However, we observe that this is not correct. Consider the example of Figure 7(a) where a new object o_5 is added. The shaded containers show the Voronoi node sets of the objects *before* o_5 is inserted. The 1NN of o_5 is the object o_3 and the only adjacent object of o_3 is o_2. However, the newly inserted object o_5 will become the 1NN of vertex q, which means the Voronoi node set of o_4 is also affected even though it is not an adjacent object of o_3 (the 1NN of the newly inserted object). In the example, the Voronoi node sets of objects o_2, o_3, and o_4 are affected by the insertion of o_5. We now describe how to determine a correct affected set of inserted object o.

Let $MaxRadius(p)$ of an object p be the maximum network distance between p and a vertex v in its Voronoi node set, i.e., $MaxRadius(p) = \arg\max_{v \in Vns(p)} d(p, v)$. Theorem 2 identifies a condition to construct the affected set.

Theorem 2. An object p is not in the affected set $A(o)$ of o if $d(o, p) \geq 2 \times MaxRadius(p)$.

Proof: We prove this by contradiction. Assume there exists a vertex $v \in Vns(p)$ for which o is the new 1NN. Since $d(o, p) \geq 2 \times MaxRadius(p)$ and $d(p, v) \leq MaxRadius(p)$, we have $d(o, p) \geq 2 \times d(p, v)$. Subtracting $d(p, v)$ on both sides gives, $d(o, p) - d(p, v) \geq d(p, v)$. By triangular inequality, $d(o, p) - d(p, v) \leq d(v, o)$. Therefore, $d(v, o) \geq d(p, v)$ and o cannot be the 1NN of v which contradicts the assumption. □



(a) Query Time (b) Update Time

Fig. 8: Handling Updates on Florida Road Network

This theorem is used to compute the affected set of o as follows. First, we find the 1NN p of o and initialize the affected set $A(o)$ with p. Then, we conduct a breadth-first search (BFS) on the adjacency graph from p. For any expanded object o_e, if it satisfies the condition in Theorem 2, it is pruned (i.e., the BFS is not expanded from o_e). Otherwise, it is included in the affected set. Note that $A(o)$ may contain some objects that are not affected, but this does not affect correctness.

$MaxRadius(p)$ for every object p can be computed essentially for free during NVD construction. Storing these values incur a small storage overhead linear to the input $O(|inv(t)|)$ (the size of the inverted list for the indexed keyword t). Also $d(o, p)$ can be conveniently computed using the Network Distance Module already available in the K-SPIN framework.

Once the affected set is computed, we perform lazy insertion of object o. Rather than inserting o into the quadtree, we insert o in the adjacency graph. Specifically, we add o to the node of each object in its affected set. For the example of Figure 7, we add o_5 to the nodes of o_2, o_3, and o_4 as shown in Figure 7(b). The nodes of the adjacency graphs are now assumed to contain one or more objects, e.g., the Heap Generator initializes the heap by inserting the 1NN of q and all the objects stored in the node. In the example, q is located in the Voronoi node set of o_4. Since o_5 was also added to the node of o_4, the heap is initialized with both o_4 and o_5.

Figure 8 shows the effect of our proposed techniques to handle updates. Specifically, we chose three keywords distributed in the lower, middle and higher thirds of the frequency distributions and the corresponding APX-NVDs are called large, medium and small, respectively. For each NVD, we inserted $x\%$ of the total objects in it using lazy updates and studied the effect of lazy updates on the query processing time in Figure 8(a). As expected the processing time has increased but the results are still impressive. In Figure 8(b), we report the average time per insertion as well as the total time to rebuild the NVD after the lazy updates. The lazy update cost is only 1ms even when 5% objects are inserted in the large NVD and the cost to rebuild NVDs is under one second. Lazy updates allow the system to continue processing of incoming queries while a new APX-NVD may be built in parallel.

Non-NVD Updates: It is possible that an NVD does not exist when inserting an object or adding a keyword to an existing object. This may occur when a keyword is new, or there are fewer than ρ objects in a keyword's inverted list. However, we do not need to construct a new APX-NVD until there are at least ρ objects plus the additional threshold for lazy updates. For deletion, however, if an APX-NVD is no longer required because there are fewer than ρ objects, then NVD updates are unnecessary, and the objects only need to be removed from the inverted list. Handling updates in the road networks (e.g., a new edge, or a deleted edge) is much more complicated and may invalidate NVDs as well as the network distance module. This is challenging for all existing techniques including shortest path algorithms and requires further research. We remark that such updates occur less frequently.

Region	$\|V\|$	$\|E\|$	$\|O\|$	$\|doc(V)\|$	$\|W\|$
DE	48,812	119,004	2,369	9,539	2,103
ME	187,315	412,352	7,827	38,590	5,289
FL	1,070,376	2,687,902	48,560	265,769	17,628
E	3,598,623	8,708,058	111,085	725,944	33084
US	23,947,347	57,708,624	688,918	3,517,112	106,559

TABLE 2: Road Network Graphs and Keyword Datasets

Parameter	Values
Road Networks	DE, ME, FL, E, **US**
No. of Results (k)	1, 5, **10**, 25, 50
No. of Terms	1, **2**, 3, 4, 5, 6

TABLE 3: Experimental Parameters (Defaults in Bold)

6.3 Improved Pre-Processing Scalability

The ideas proposed in Sections 6.1 and 6.2 combine in a elegant way to address the pre-processing woes described in Section 5. Utilizing ρ-Approximate NVDs instead of exact NVDs reduce the space requirement by more than an order of magnitude. Exploiting the Zipfian nature of keywords to eliminate keyword indexes for a vast majority of them substantially reduces the construction time. For example, the 90GB index size and 3-day build time for exact NVDs for the US road network dataset is reduced dramatically to 584MB and 1.5 hours, respectively. Additionally, updating approximate NVDs for changes to objects is considerably cheaper. Finally, these benefits incur only a small bounded penalty in query performance while still returning exact results.

7 Experiment Results

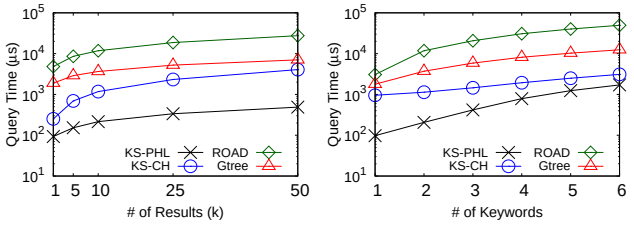
7.1 Experimental Setting

Competing Methods: We compare three variants of K-SPIN. All variants use our ρ-Approximate Network Voronoi Diagram (Section 6.1) for the Keyword Separated Index and the ALT index [15] for the Lower Bounding Module. ALT computes lower-bound network distances between any two vertices using pre-computed distances to "landmark" vertices and the triangle inequality. The variants differ in their Network Distance Module. KS-CH employs Contraction Hierarchies (CH) [10], KS-GT uses G-tree [4] and KS-PHL utilizes Pruned Highway Labeling (PHL) [11]. Each index offers a different trade-off between pre-processing cost and query performance. Generally speaking, PHL offers fast queries at the expense of high space cost, while CH offers considerably less space cost but relatively slower queries.

K-SPIN answers both top-k and BkNN queries (Section 4) and returns *exact* results. We compare with state-of-the-art top-k techniques ROAD [3] and G-tree [4], and BkNN technique FS-FBS [2]. We also adapt G-tree to answer BkNN queries. We exclude network expansion methods as past results [2] (that we verified) showed them to be orders of magnitude slower. Note that G-tree can also answer network distance queries.

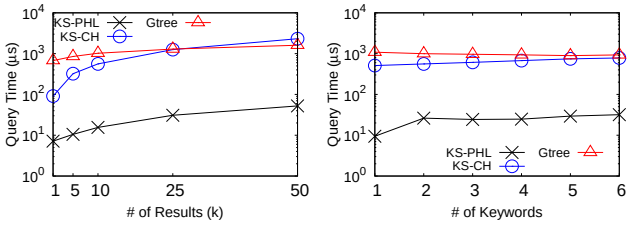
We obtained code for G-tree and ROAD from [23]. We modified it for spatial keywords and implemented the query algorithms. The code for PHL and FS-FBS was obtained from the authors. The parameters for the G-tree, FS-FBS and ROAD indexes were chosen as in past studies [2], [4], [12] for best query performance with practicable index construction.

Environment: We conduct experiments on a Linux (64-bit) dedicated Amazon Web Services c4.8xlarge instance with two



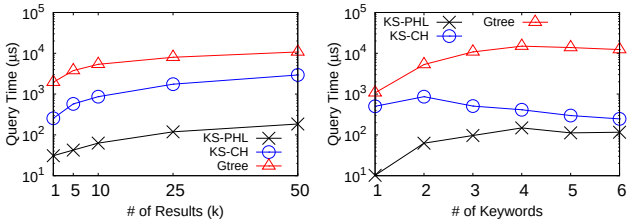
(a) Varying k (b) Varying # of Keywords

Fig. 9: Top-k Queries (US,# of terms=2, k=10)



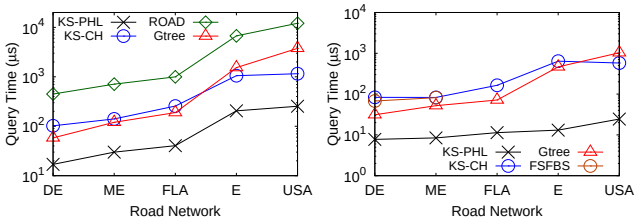
(a) Varying k (b) Varying # of Keywords

Fig. 10: Disjunctive BkNN (US,# of terms=2,k=10)



(a) Varying k (b) Varying # of Keywords

Fig. 11: Conjunctive BkNN (US,# of terms=2,k=10)



(a) Top-k (b) BkNN (Disjunctive)

Fig. 12: Varying Road Network (# of terms=2, k=10)

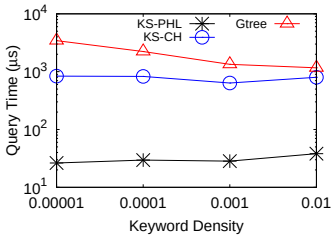
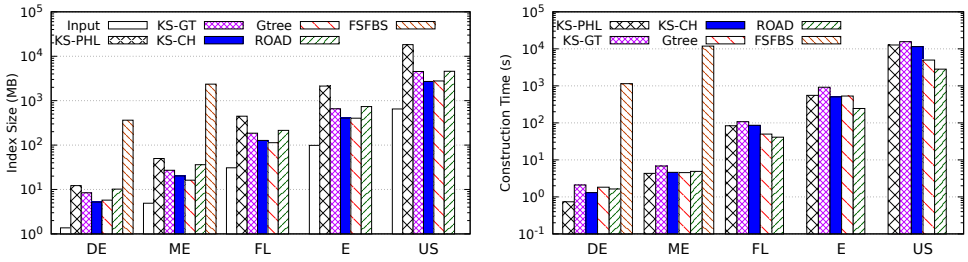


Fig. 13: Varying Frequency BkNN (US,# of terms=1, k=10)



(a) Index Size (b) Construction Time

Fig. 14: Index Pre-Processing Time and Space

Intel Xeon E5-2666v3 2.9GHz 10-core CPUs and 60GB DDR4-1866 memory. Code was written in C++ and compiled by g++ v5.4 with O3 flag. Query algorithms use a single thread. All experiments were conducted using memory-resident indexes. This setting is preferred given the high query throughput demands and viable given the affordability of RAM. This was very apparent when a disk-based variant of FS-FBS performed slower than Dijkstra's algorithm using only the input graph in memory [2].

Datasets: We used five real-world road network graphs as listed in Table 2. The DE (Delaware), ME (Maine), FL (Florida), E (Eastern United States), and US (United States) datasets were created for the 9th DIMACS Challenge [24] and used widely in recent studies [4], [23]. We extracted points of interest (POIs) and their descriptors from OpenStreetMap (OSM) [25]. Each POI was mapped to the closest road network vertex and keywords were extracted from its descriptors. Table 2 lists the statistics for the keyword dataset of each road network, where $|O|$ is the number of object vertices (POIs), $|W|$ is the number of unique keywords, and $|doc(V)|$ is the number of keyword occurrences in all objects.

Query Parameters: We investigate the effect of varying (a) number of results k, (b) number query keywords, (c) dataset size, and (d) keyword frequency. Table 3 lists the parameter values with defaults in bold. We create a set of query keyword vectors by first choosing several popular search terms including "hotel", "restaurant", "supermarket", "bank", and "school". For each term, we select an object o that contains the keyword. We select further keywords associated with o to create query keyword vectors of length 1 to 6. This ensures that combinations of query keywords are correlated because they exist for a real-world object and is similar to the process used in a recent spatial keyword experimental study [8]. We repeat this until we have selected 10 objects for each of the five terms, generating a total 50 vectors for each length. Each vector is combined with 100 uniformly selected query vertices for a total of 5,000 queries over which we report the average query time. The query time for K-SPIN includes creation and maintenance of the on-demand inverted heaps.

7.2 Query Performance

Top-k Queries: For increasing k (Figure 9(a)) and numbers of query keywords (Figure 9(b)) on the US dataset, both K-SPIN methods significantly outperform the next best competitor by at least several times on all settings. KS-PHL, in particular performs, up to several orders of magnitude faster, demonstrating the advantages of K-SPIN's modular nature by allowing the faster network distance technique, PHL, to be used. While the performance gap between KS-CH over G-tree is consistent in Figure 9(b), the performance gap between KS-PHL and other methods decreases with additional keywords. PHL is a significantly faster network distance method than CH. As a result, with increasing keywords, the cost of maintaining additional inverted heaps takes a bigger proportion of the query time in KS-PHL. But, in real terms, both KS-PHL and KS-CH are increasing by the same margin despite appearing otherwise due to the logarithmic scale, thus KS-PHL will never "catch-up" to KS-CH. Note that all K-SPIN query times in all experiments include the cost of lazy heap initialization and maintenance (described in Algorithm 4 and in Section 6).

Boolean kNN Disjunctive Queries: Figure 10 shows the query performance for disjunctive BkNN queries. KS-PHL again signif-

icantly outperforms the other techniques irrespective of k or the number of keywords. Interestingly, KS-CH does not improve over G-tree as significantly as for top-k queries in some cases, e.g., for $k = 50$ in Figure 10(a). The reason for this is two-fold. First, disjunctive queries are easier to answer, in a sense, than top-k and conjunctive queries because the criteria only requires an object to have any one query keyword. Thus, in general, we can expect result objects to be found closer to the query location, which means they appear in nearby G-tree nodes that are less costly to evaluate. This also explains why G-tree improves marginally with increasing query keywords (objects are easier to match). Second, G-tree is able to re-use intermediate network distance computations, hence scales efficiently for increasing k because many of them can be re-used. However, we note that a similar strategy has been applied to Dijkstra-based hierarchical methods in the past [26] by saving and re-using the forward search between network distance computations which may also be applied to CH. Nonetheless, we note that KS-CH uses less memory than G-tree but is still able to match or beat its performance on disjunctive queries without this improvement. FS-FBS is not shown as it cannot be built for the large US dataset.

Boolean kNN Conjunctive Queries: Figure 11 depicts performance on conjunctive BkNN queries. The advantage of K-SPIN methods over G-tree is even more pronounced than disjunctive queries, e.g., with several times to orders of magnitude improvement for varying k in Figure 11(a). Keyword aggregation used by G-tree is more susceptible to false positives for *conjunctive* queries as the hierarchy must be evaluated deeper before false positives can be identified. K-SPIN on the other hand can quickly eliminate objects not satisfying the criteria, avoiding computation of expensive network distances. We also see that increasing query keywords results in improving query times for K-SPIN methods in Figure 11(b). With additional keywords, there are fewer objects that match the conjunctive criteria. Consequently, the least frequent keyword is more likely to have an even lower frequency. This gives K-SPIN an advantage as it has to consider fewer candidates (i.e., only those that contain the least frequent keyword), explaining the observed improvement.

Varying Road Network: Figures 12(a) and 12(b) depicts the query time of each technique for top-k queries and disjunctive BkNN queries, resp., for varying road network size. The number of vertices in the network increases from left to right. First, KS-PHL significantly outperforms the other techniques on all datasets for both types of queries. Second, we generally see that the performance improvement of K-SPIN techniques over competing methods increase with dataset size. This shows keyword separation scales better with dataset size as the occurrence of false positives is reduced. This can be explained by the fact that, in a bigger graph, higher levels of the G-tree and ROAD hierarchies aggregate more keyword occurrences. This results in degraded pruning power and hence more false positives and redundant network distance computations in G-tree and ROAD.

Varying Keyword Frequency: Figure 13 illustrates the effect of increasing keyword frequency. We express frequency in terms of keyword object density $|inv(t)|/|V|$, where $|inv(t)|$ is the number of objects which contain keyword t and $|V|$ is the total number of vertices in the road network. Each tic on the x-axis represents a "bucket" of keywords in the density range greater than or equal to the current tic but less than the next tic (the last tic includes keywords of all densities larger than 0.01). We execute single-keyword BkNN queries to isolate the impact of frequency. Once again K-SPIN outperforms G-tree, with KS-PHL more than an order of magnitude faster. KS-CH improvement over G-tree is smaller as only a single query keyword is involved, allowing G-tree to avoid the false positive problems seen earlier with the more realistic multi-keyword disjunctive BkNN and top-k queries.

7.3 Index Performance

Figure 14(a) shows the size of each index. "Input" is the input graph and keyword dataset. Contraction Hierarchies entails the smallest footprint out of all indexes at 2.6GB compared to 2.8GB for G-tree on the largest dataset (US). KS-PHL entails an index size of 17.9GB compared to 4.5GB for ROAD for the US. The FS-FBS index could only be constructed for the two smallest datasets. The 2-hop labeling index used to build the FS-FBS index requires a node order. As described in the original study [2], we tested several node orders generated by Contraction Hierarchies [10] (including reverse order), but could not build an index for FL in less than 24-hours, not to mention the prohibitive scaling of index size. Unlike K-SPIN, FS-FBS does not provide an easy way to replace the road network index used. Apart from FS-FBS, the pre-processing time of each technique in Figure 14(b) is comparable. K-SPIN received a useful speed-up from parallelization (Section 6.1), while other techniques cannot be as easily parallelized.

7.4 False Positive Performance

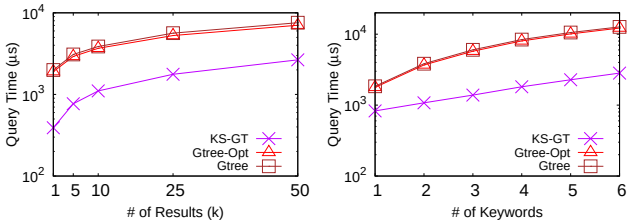
In Section 1, we presented examples of how existing spatial keyword algorithms that use keyword aggregation, like G-tree, incur costly additional work due to false positive candidates. So far we have shown empirical evidence that shows K-SPIN outperforms its keyword aggregation counterparts, often quite significantly. To give further credence to our claim that K-SPIN does indeed reduce the occurrence of false positives, we present a deep-dive experimental comparison using the G-tree index.

We use the G-tree index as the network distance module in K-SPIN (referred to as KS-GT). So KS-GT and G-tree's spatial keyword query algorithms use the same underlying road network index (G-tree) to compute network distances. This occurs in exactly the same manner, e.g., already computed partial network distances are re-used for later computations, described as *materialization* by Zhong *et al.* [4]. As a result, we perform an apples-to-apples comparison and thus obtain a truer understanding of the reduction in false positives achieved by K-SPIN. Before presenting our findings, we first describe how to apply keyword separation principles to G-tree's spatial keyword query algorithms using.

7.4.1 Applying Keyword Separation Principles to G-tree

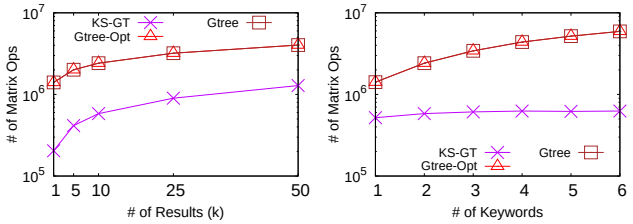
Given the disadvantages of keyword aggregation and advantages of keyword separation described so far, two pertinent follow-up question may arise: (1) can principles of keyword separation be applied to existing spatial keyword algorithms and (2) does applying such principles mitigate the drawbacks of keyword aggregation discussed earlier. We answer (2) in the subsequent section (the short answer being "no"), but first describe how an answer to (1) can be implemented for the top-k query algorithm proposed by Zhong *et al.* [4] using the G-tree index.

G-tree is a tree data structure where each tree node represents a road network subgraph. Starting with the entire road network as the root, each child node is a partitioning of the parent node's subgraph. G-tree's top-k algorithm finds candidates by traversing



(a) Varying k (b) Varying # of Keywords

Fig. 15: Top-k Query Time (US,# of terms=2,k=10)



(a) Varying k (b) Varying # of Keywords

Fig. 16: Top-k Matrix Operations (US,# of terms=2,k=10)

this subgraph hierarchy up from the leaf node containing the query and down towards other leaf nodes containing objects. Each tree node is associated with an *occurrence list*, which lists all child nodes with an object. Occurrence lists can then be used to avoid searching G-tree nodes (i.e., subgraphs) without objects. In the case of top-k queries, the set of objects consists of all vertices that contain a keyword. Thus, each tree node's occurrence list indicates which child nodes contain an object with *any* keyword. Note that this is in addition to the pseudo-document associated with each tree node, which contains all the keywords present in the subgraph, as usual for the keyword aggregation approach. By using an occurrence list, child nodes without objects can be pruned immediately without consulting their pseudo-documents.

We observe that keyword separation principles can be applied to occurrence lists. Rather than building one occurrence list for a tree node, build a separate occurrence list for each keyword in the tree node's pseudo-document. Now G-tree's top-k algorithm can be modified to prune child nodes that do not contain objects with any of the query keywords. Note that we use this optimized version of G-tree spatial keyword algorithms in all previous experiments.

7.4.2 Query Time and Network Distance Cost

Figure 15 displays the query time of KS-GT, optimized G-tree described above (Gtree-Opt), and G-tree's original top-k query. Additionally, we compare methods in terms of *matrix operations* in Figure 16. Computing network distance using G-tree involves determining the tree path between source and destination vertices in the G-tree hierarchy. Given this path, distances are computed to each border associated with a tree node on the path by looking up and summing distance matrix elements (described in detail by Zhong *et al.* [4]). We term this look-up and sum as a machine-independent *matrix operation* that accurately captures how costly the network distance was to compute. Most importantly, if fewer false positives occur there will be fewer matrix operations.

In Figure 15 we see that Gtree-Opt marginally improves on the original G-tree top-k query algorithm in terms of query time. However, in Figure 16 we see little to no improvement in terms matrix operations. This suggests that the query time improvement is entirely from avoiding pseudo-document look-ups rather than incurring fewer false positives. Identical numbers of matrix operations shows that the hierarchy is still being evaluated to the same depth to overcome the effect of aggregation. These observations strongly evince that problems arising from keyword aggregation cannot be easily solved in existing techniques.

In Figure 15, KS-GT consistently outperforms G-tree by up to an order of magnitude in terms of query time. This is despite KS-GT query time including extra overheads, e.g., computing lower-bounds and initializing/maintaining inverted heaps. The even greater improvement on matrix operations in Figure 16 removes any doubt. The improvement in matrix operations directly shows that K-SPIN utilizes the G-tree index more efficiently, i.e., due to fewer false positives. We cannot apply further keyword separation to G-tree itself due to the permanent loss of discriminating information without reversing the keyword aggregation itself. K-SPIN in fact achieves this, but in a simple and versatile manner.

8 Related Work

Road Network Top-k Queries: Similar to G-tree, ROAD is a hierarchical partitioning of the road network. ROAD was originally developed to answer kNN queries (i.e., not involving keywords) [12]. When applied to top-k spatial keyword queries [3], ROAD experiences the exact same problematic scenarios that G-tree does as presented in Section 1.1. The difference between ROAD and G-tree is largely the way the subgraph hierarchy is stored and accessed. G-tree offers better cache performance, and hence query times, compared to ROAD [23].

Road Network Boolean kNN Queries: FS-FBS [2] is a Boolean kNN query technique improving on earlier approximate techniques [27] by providing exact results. The related LARC [28] is an adaptation for continuous queries. FS-FBS uses a 2-hop labeling index and its inverse, a backwards label index. In a 2-hop labeling index, each vertex has a *label*, a set of *hub* vertices to which distances are known. It is guaranteed that any two vertices have a common hub that lies on the shortest path between them. A backwards label for hub h is the list of vertices for which h is in their label. For frequent keywords, FS-FBS employs keyword aggregation on backwards labels using a bit-array hash to indicate presence of keywords. It faces familiar keyword aggregation problems as the bit-array hash incurs collisions, leading to false positives and unnecessary distance computations. For infrequent keywords, FS-FBS simply computes network distances to *all* vertices containing the infrequent keyword. First, it is problematic to differentiate between frequent and infrequent keywords. While a metric is suggested, it is still necessary to verify the best performing frequency experimentally [2]. Second, inverted lists for infrequent keywords, which are more common, cannot be accessed by FS-FBS in order. Thus it is not possible to terminate without evaluating the entire list. In contrast, on-demand inverted heaps offer cheap ordered access and terminate sooner.

Other Queries: Spatial keyword queries have been widely studied in Euclidean space [6], [7], [8], [9], where Euclidean distance is the metric for spatial proximity in computing spatio-textual scores. The downside is that other useful metrics such as travel time are not supported. k-Nearest Neighbor queries in road networks [4], [12], [16], [29] also search for nearby points of interest using road network distance. However, such queries are executed on pre-determined object sets, requiring POIs to be sanitized and categorized. They also do not take keywords or spatio-textual scores into account. While network distances

in inverted heaps increase steadily (as objects are further away from the query), the same cannot be expected of inverted lists sorted by keyword impact. This, and the fact that only lower-bound distances are available, means more false positives if using the Thresholding Algorithm (TA) [30], making it unsuitable for top-k spatial keyword queries. TA also only supports monotonic scoring functions, eliminating weighted distance.

9 Conclusions

Keyword separation is a viable alternative to keyword aggregation, as evident in the significant improvement in query performance of K-SPIN over competing methods. This holds true particularly given the improvement of K-SPIN using G-tree over G-tree's own query algorithms. Moreover, this need not come at a prohibitive pre-processing cost, as shown by the substantial reduction in keyword index size and time. In fact, utilizing the long-tail of Zipfian distributions and ρ-Approximate NVDs are useful techniques on their own. Ultimately, K-SPIN provides an efficient and versatile framework for spatial keyword query processing, in addition to provision for dynamic updates and parallelized index building.

Acknowledgments

We sincerely thank Hanan Samet for his insightful comments. The research of Muhammad Aamir Cheema is supported by ARC DP180103411 and FT180100140. Arijit Khan is supported by MOE Tier-1 RG83/16 and NTU M4081678. Tenindra Abeywickrama is supported by an Australian Government RTP Scholarship.

References

[1] G. Cong and C. S. Jensen, "Querying Geo-Textual Data: Spatial Keyword Queries and Beyond," in *SIGMOD*, 2016, pp. 2207–2212.
[2] M. Jiang, A. W.-C. Fu, and R. C.-W. Wong, "Exact Top-k Nearest Keyword Search in Large Networks," in *SIGMOD*, 2015, pp. 393–404.
[3] J. B. R.-Junior and K. Nørvåg, "Top-k Spatial Keyword Queries on Road Networks," in *EDBT*, 2012, pp. 168–179.
[4] R. Zhong, G. Li, K. Tan, L. Zhou, and Z. Gong, "G-Tree: An Efficient and Scalable Index for Spatial Search on Road Networks," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 8, pp. 2175–2189, 2015.
[5] G. Sterling. (2015) http://screenwerk.com/2015/05/11/data-suggest-that-local-intent-queries-nearly-half-of-all-search-volume/.
[6] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial web objects," *PVLDB*, vol. 2, no. 1, pp. 337–348, 2009.
[7] D. Wu, G. Cong, and C. S. Jensen, "A Framework for Efficient Spatial Web Object Retrieval," *VLDB J.*, vol. 21, no. 6, pp. 797–822, 2012.
[8] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial Keyword Query Processing: An Experimental Evaluation," in *PVLDB*, 2013, pp. 217–228.
[9] D. Zhang, C.-Y. Chan, and K.-L. Tan, "Processing Spatial Keyword Query As a Top-k Aggregation Query," in *SIGIR*, 2014, pp. 355–364.
[10] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks," in *WEA*, 2008, pp. 319–333.
[11] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata, "Fast Shortest-path Distance Queries on Road Networks by Pruned Highway Labeling," in *ALENEX*, 2014, pp. 147–154.
[12] K. C. K. Lee, L. W.-Chien, Z. Baihua, and T. Yuan, "ROAD: A New Spatial Object Search Framework for Road Networks," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 3, pp. 547–560, 2012.
[13] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong, "Efficient Continuously Moving Top-k Spatial Keyword Query Processing," in *ICDE*, 2011.
[14] J. Zobel and A. Moffat, "Inverted Files for Text Search Engines," *ACM Comput. Surv.*, vol. 38, no. 2, 2006.
[15] A. V. Goldberg and C. Harrelson, "Computing the Shortest Path: A* Search Meets Graph Theory," in *SODA*, 2005, pp. 156–165.
[16] T. Abeywickrama and M. A. Cheema, "Efficient Landmark-Based Candidate Generation for kNN Queries on Road Networks," in *DASFAA*, 2017, pp. 425–440.
[17] R. Zhong, G. Li, K. Tan, and L. Zhou, "G-tree: An Efficient Index for KNN Search on Road Networks," in *CIKM*, 2013, pp. 39–48.
[18] M. Kolahdouzan and C. Shahabi, "Voronoi-based K Nearest Neighbor Search for Spatial Network Databases," in *VLDB*, 2004, pp. 840–851.
[19] M. Erwig and F. Hagen, "The Graph Voronoi Diagram with Applications," *Networks*, vol. 36, pp. 156–163, 2000.
[20] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck, "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms," in *SODA*, 2010.
[21] J. Sankaranarayanan, H. Alborzi, and H. Samet, "Efficient Query Processing on Spatial Networks," in *GIS*, 2005.
[22] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2005.
[23] T. Abeywickrama, M. A. Cheema, and D. Taniar, "K-nearest neighbors on road networks: A journey in experimentation and in-memory implementation," *PVLDB*, vol. 9, no. 6, pp. 492–503, 2016.
[24] http://www.dis.uniroma1.it/%7Echallenge9/.
[25] http://www.openstreetmap.org.
[26] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner, "Computing many-to-many shortest paths using highway hierarchies," in *ALENEX*, 2007, pp. 36–45.
[27] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian, "Top-k nearest keyword search on large graphs," *PVLDB*, vol. 6, no. 10, pp. 901–912, 2013.
[28] B. Zheng, K. Zheng, X. Xiao, H. Su, H. Yin, X. Zhou, and G. Li, "Keyword-Aware Continuous kNN Query on Road Networks," in *ICDE*, 2016, pp. 871–882.
[29] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable Network Distance Browsing in Spatial Databases," in *SIGMOD*, 2008.
[30] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," in *PODS*, 2001.

Tenindra Abeywickrama Tenindra Abeywickrama is presently a PhD student in the Faculty of Information Technology at Monash University, Australia. He received a B.Eng. (Electrical) and a B.Sc. (Computer Science) from UNSW Australia in 2010. Tenindra received the Cheung Kong Research Fellowship in 2018. His current research focus is on developing indexing and query processing techniques for spatial networks.

Muhammad Aamir Cheema is a Senior Lecturer at Clayton School of Information Technology, Monash University, Australia. He obtained his PhD from UNSW Australia in 2011. He is the recipient of 2012 Malcolm Chaikin Prize for Research Excellence in Engineering, 2013 Discovery Early Career Researcher Award, 2014 Dean's Award for Excellence in Research by an Early Career Researcher. His PhD thesis was nominated for SIGMOD Jim Gray Doctoral Dissertation Award and ACM Doctoral Dissertation Competition. He has won two CiSRA best research paper awards (in 2009 and 2010), two invited papers in the special issue of IEEE TKDE on the best papers of ICDE (2010 and 2012), and two best paper awards at WISE 2013 and ADC 2010, respectively. He served as PC co-chair for ADC 2015, ADC 2016, 8th ACM SIGSPATIAL Workshop ISA 2016, WWW International Workshop on Social Computing 2017, proceedings chair for DASFAA 2015, tutorial co-chair for APWeb 2017 and publicity co-chair for ACM SIGSPATIAL 2017.

Arijit Khan Arijit Khan is an Assistant Professor at Nanyang Technological University, Singapore. He earned his PhD from the University of California, Santa Barbara, and did a post-doc in the Systems group at ETH Zurich. Arijit is the recipient of the IBM PhD Fellowship in 2012-13. He co-presented tutorials on graph queries and systems at ICDE 2012, VLDB 2014, 2015, 2017. His research interests include big-graphs management and analytics, with a focus on user-friendly, online querying and pattern mining in social and information networks, using scalable algorithms and machine learning techniques.