U-ASK: A Unified Architecture for kNN Spatial-Keyword Queries Supporting Negative Keyword Predicates

Yongyi Liu
University of California, Riverside
Riverside, California
yliu786@ucr.edu

Amr Magdy*
University of California, Riverside
Riverside, California
amr@cs.ucr.edu

ABSTRACT

Spatial keyword queries have been popular in the research community for over a decade due to the explosive growth in user-generated data and its prime applications in different domains. kNN queries make a major category of spatial keyword queries that is heavily studied. However, the expressiveness of existing kNN queries is limited in supporting negative keyword predicates, e.g., *find tweets with keywords "Chipotle" but NOT "Chipotle sauce"*, which have prime applications. In addition, existing architectures suffer from a lack of generality for different types of kNN queries. This paper proposes *U-ASK*; a Unified Architecture for Spatial-Keyword query supporting negative keyword predicates. *U-ASK* includes an indexing framework named TEQ (Textual-Enhanced Quadtree) and a query processor POWER (Parallel bOttom-up search With incrEmental pRuning) that handle various forms of kNN spatial keyword queries with negative keyword predicates. The experimental evaluation on real tweet datasets demonstrates up to 30× faster runtime compared to the state-of-the-art algorithms.

CCS CONCEPTS

• **Information systems → Top-k retrieval in databases**.

KEYWORDS

Spatial-keyword Query Processing, Spatial-Textual Indexing

ACM Reference Format:
Yongyi Liu and Amr Magdy. 2022. U-ASK: A Unified Architecture for kNN Spatial-Keyword Queries Supporting Negative Keyword Predicates. In *The 30th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '22), November 1–4, 2022, Seattle, WA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3557915.3560975

1 INTRODUCTION

The rich applications of spatio-textual objects have motivated immense research on spatial-keyword search to build efficient indexing frameworks and low-latency query processing algorithms [3, 4, 6, 14, 16, 17, 20, 24–26, 28, 32, 35, 36, 39–43]. Spatio-textual objects, i.e., objects with both spatial and textual descriptions, are being generated in numerous numbers due to the advancement of geo-tagging techniques and social networking services. Examples of such objects include Points of Interest (POIs), e.g., Steakhouse in Manhattan, NYC, social media posts, e.g., geo-tagged tweets from Twitter. Such objects are being utilized in applications as disparate as POIs search [7, 27, 45], analyzing users' preference [44, 47], identifying trending topics [1, 30, 31, 34], crime prevention [18], and foodborne illness analysis [33].

A major category of spatial-keyword queries includes kNN queries that are two types [8]: (1) Top-k kNN queries that rank objects based on both spatial and textual relevance to query parameters, and (2) Boolean kNN queries that rank objects based on only spatial distance. In both cases, kNN queries involve multiple keywords that are combined through two famous conjunctions: (a) *AND*, e.g., *find tweets near Riverside, CA that mention "fire" AND "evacuation"*, and (b) *OR*, e.g., *find tweets near Riverside, CA that mention "party" OR "painting"*. However, such conjunctions are not enough to handle all the potential applications of spatial-keyword queries. For example, in our collaboration with food scientists on analyzing social media data to early signal foodborne illness outbreaks using Chipotle Mexican Grill restaurants as a use case [2], we obviously suffered from noisy results that deteriorated the whole analysis. In specific, posting spatial-keyword queries with keywords such as *"Chipotle"* and *sick* to filter out relevant tweets generates immense noise in the results that mention phrases such as *"chipotle sauce"*, *"that's sick"*, *"sick of this show"*, etc. Such noise has represented the majority of the results, so we ended up putting significant human efforts in filtering out relevant data to provide a high-quality analysis.

To enrich the filtering capabilities of spatial keyword search, supporting negative keyword predicates is essential to provide customized search with high flexibility. For the above example, we need to post queries that find tweets with keywords *"Chipotle"* but *NOT "chipotle sauce"*, and queries that find tweets with keywords *"sick"* but *NOT "that's sick"* or *"sick of"*, for instance. Such queries cannot be supported using existing techniques. To the best of our knowledge, negative keyword predicates are only supported in [5]. However, the work in [5] has two limitations. First, the keywords in negative keyword predicates are considered a bag of words, meaning it cannot deal with phrases with a sequence of words such as *"chipotle sauce"* or *"that's sick"*. Instead, *"chipotle" "sauce"* are considered as two separate words, which is not semantically correct in our motivating application and produces misleading results. Second, [5] supports only Boolean kNN spatial-keyword

*This work is partially supported by the National Science Foundation, USA, under grants IIS-1849971, SES-1831615, and CNS-2031418.

SIGSPATIAL '22, November 1–4, 2022, Seattle, WA, USA

ACM ISBN 978-1-4503-9529-8/22/11.
https://doi.org/10.1145/3557915.3560975

queries, so their proposed model cannot be generalized to support top-k kNN spatial-keyword queries, unlike our work that supports the two types of kNN queries.

This paper proposes *U-ASK*; a unified architecture to support the two types of kNN spatial-keyword queries with different types of conjunctions: *AND*, *OR*, and *NOT* conjunctions, all in a single framework. Therefore, *U-ASK* allows the system administrator to use a single module to support the two types of kNN spatial-keyword queries with flexible conjunctions. This is highly favorable from a system builder perspective to maintain a single system asset that supports a variety of queries efficiently. *U-ASK* consists of an indexing framework TEQ (Textual-Enhanced Quadtree) and a query processing algorithm POWER (Parallel bOttom-up search With incrEmental pRuning) with its variants to serve various kNN spatial-keyword queries with flexible constraints. TEQ provides an efficient hybrid index for spatial-textual objects where the quadtree structure is used to index objects spatially and the inverted textual index is used to index the keywords. With TEQ, the query processor can efficiently and precisely locate the objects with great similarity to the query by first identifying the cell with great spatial proximity and then accessing the textual inverted lists corresponding to the query keywords. POWER and its variants utilize the TEQ index to perform parallel spatial-keyword queries with low latency. Furthermore, to the best of our knowledge, POWER and its variants are the first kNN spatial-keyword query processing techniques that harness the full potential of multi-core parallelization to speed up the search. The reason is that existing major query processing algorithms have a strong dependency among different steps, which makes them hard to parallelize. The experimental results show that our algorithm provides up to 30× faster compared to the state-of-the-art techniques. Our contributions are summarized as follows:

- We propose two novel kNN spatial-keyword query problems: Top-k kNN Query with Negative keyword predicates (TKQN), and Boolean kNN Query with negative keyword predicates (BKQN) .
- We introduce a Unified Architecture for Spatial-Keyword query with negative keyword predicates (*U-ASK*) that includes indexing components TEQ, and query processing algorithms POWER with its variants to handle TKQN and BKQN queries.
- We conduct extensive experimental evaluation on real datasets that shows the superiority of our techniques compared to the state-of-the-art literature.

The rest of this paper is organized as follows: Section 2 presents the related work. Section 3 formally defines the problem. Sections 4 details the TEQ indexing framework. Section 5 demonstrates the query processing algorithms for TKQN and BKQN. Section 6 presents the experimental evaluation and Section 7 concludes the paper.

2 RELATED WORK

Spatial-keyword queries are heavily studied in the literature in two categories: fundamental spatial keyword queries [12–16, 23, 24, 28, 29, 36, 38, 41–43] and variations of fundamental spatial keyword queries [3, 4, 6, 9, 11, 17, 20, 25, 26, 32, 35, 39, 40, 46]. Fundamental spatial-keyword queries include kNN queries [14, 16, 24, 28, 36, 41–43], which return the k objects that are the most relevant to the query, and range queries [12, 13, 15, 23, 29, 38], which return all the objects satisfying the query constraints within a specific spatial range. Variations of fundamental queries include moving spatial keyword queries [25, 39, 40], group spatial-keyword queries [3, 4, 6, 17, 20, 32], and spatial skyline queries [26, 35]. Our focus in this paper is kNN queries. Existing kNN spatial-keyword queries can be categorized into top-k kNN queries and Boolean kNN queries depending on how the spatial and textual attributes are involved in the top-k ranking.

Boolean kNN Queries [14, 16, 24, 24, 28, 36, 41–43] rank the spatio-textual objects based on the spatial distance between the objects and the query and the textual predicate serves as a Boolean filter. The textual predicates could be based on *AND* conjunction [16, 24, 36, 41, 43] where each resulting object must include all the query keywords, or *OR* conjunction [43] where each resulting object must include at least one query keyword. Felipe et al. proposed IR^2-Tree [16] which incorporates textual signature, i.e., keyword bitmap, into R-Tree [22] nodes for efficient filtering. Tao et al. introduced Spatial Inverted Index (SI-index) [36] that builds an R-Tree for each textual inverted list with a low storage cost. IL-QuadTree proposed in [41] builds a linear quadtree [21] for each keyword and searches multiple trees during the query. Hong et al. [24] proposed an optimization based on IL-QuadTree that requires traversing only one quadtree, regardless of the number of query keywords, by assigning a priority score to each keyword and visiting the tree corresponding to the query keyword that has the highest priority. Cary et al. [5] proposed Spatial Keyword Index (SKI) that could support *AND*, *OR*, and *NOT* conjunctions at the same time by the efficient usage of bitmap. However, it cannot handle the negative keyword phrases as detailed before.

Top-k kNN Queries [14, 24, 28, 41, 42] rank the spatio-textual objects based on both spatial and textual relevance to the query parameters. Compared to Boolean kNN query, top-k kNN query is more flexible because the importance of spatial proximity and textual proximity could be adjusted by a weighting factor in different applications. Cong et al. [14] proposed IR-Tree, which equips each node in the tree with aggregate textual information and then applies a top-down best-first search to fetch the top-k results. Zhang et al. proposed RCA [42] that models the spatio-keyword query as a top-k aggregation query and applies inverted index and Z-order mapping to index textual and spatial dimensions, respectively.

Distinguished from all existing work, our work efficiently process both Boolean kNN query and top-k kNN query. Moreover, we incorporate negative keyword predicates into the above two types of queries to provide flexible queries and enrich their filtering capabilities to serve a wide variety of applications.

3 PROBLEM DEFINITION

In this section, we formally define Top-k kNN Query with Negative keyword predicate (TKQN) and Boolean kNN Query with Negative keyword predicate (BKQN).

A spatio-textual object o is a triple $o = (o.id, o.loc, o.txt)$ where $o.id$ is the unique identifier of the object, $o.loc$ is the location of o,

i,e, latitude and longitude, and $o.txt$ is the textual description of o that consists of a sequence of strings. We refer the weight of a keyword $w \in o.txt$, $s(o.w)$, as the relative keyword frequency of w, which is computed as the count of w in $o.txt$ divided by the size of $o.txt$. $s(o.w)$ can also be computed using other measures such as TF-IDF, which does not affect our query processing as long as the weight of a keyword in an object does not change with changing the query parameters and remains static in the database. We will refer to spatio-textual object as object when there is no ambiguity.

Top-k kNN with Negative keyword predicates (TKQN) query q_t is a quintuple:

$$q_t = (q_t.loc, q_t.pos, q_t.neg, q_t.\lambda, q_t.k)$$

Where $q_t.loc$ is the location, i.e., latitude and longitude, of the query, $q_t.pos$ is a set of positive keywords, $q_t.neg$ is a set of negative phrases, each is a sequence of one or more keywords, $q_t.\lambda$ is a weighting factor, $\lambda \in [0, 1]$, that adjusts the importance of spatial proximity and textual proximity of the query results, and $q_t.k$ is an integer. Additionally, we use $q_t.negLen$ to denote the average length of the negative phrases in $q_t.neg$.

Query q_t returns k objects o_i, $1 \leq i \leq k$, so that: (1) $o_i.txt \bigcap q_t.pos \neq \emptyset$, i.e., o_i includes at least one query positive keywords, (2) $q_t.negPh \nsubseteq o_i.txt \ \forall q_t.negPh \in q_t.neg$, i.e., o_i does not include any of the negative phrases, and (3) o_i is top-k ranked according to $score(o_i, q_t)$ that combines the spatial and textual relevance of o_i to q_t. Formally,

$$score(o_i, q_t) = q_t.\lambda * score_s(o_i, q_t) + (1 - q_t.\lambda) * score_t(o_i, q_t) \quad (1)$$

$score_s(o_i, q_t)$ measures the spatial proximity between o_i and q_t, which is computed as one minus the Euclidean distance between o_i and q_t divided by the maximum pair-wise distance in the space, i.e., $dist_{Max}$. Formally,

$$score_s(o_i, q_t) = 1 - \frac{dist(o_i, q_t)}{dist_{Max}}$$

$score_t$ is the textual proximity between o_i and q_t and it is computed as the sum of the weights of all keywords $w \in \{o_i.txt \bigcap q_t.pos\}$. Formally,

$$score_t(o_i, q_t) = \sum_{w \in o_i.txt \bigcap q_t.pos} s(o_i.w)$$

Example. Table 1 shows examples of spatio-textual objects. Suppose q_t is a TKQN instance with $q_t.loc = (36.95, -120.89)$, $q_t.pos = \{Chipotle\}$, $q_t.neg = \{[Chipotle\ sauce], [Chipotle\ grill]\}$, $q_t.\lambda = 0.5$, $q_t.k = 1$. $o2$ and $o4$ cannot be valid candidates for q_t because they include either *Chipotle sauce* or *Chipotle grill*. $o3$ and $o5$ cannot be a valid candidate as well because they do not include *Chipotle*. The remaining objects are $o1$ and $o6$ and they both have the same textual proximity $\frac{1}{6}$. $o6$ will be returned as the result because $o6$ is closer to q_t, and thus has a higher spatial proximity.

Boolean kNN Query with Negative keyword predicates (BKQN) query q_b is a quintuple:

$$q_b = (q_b.loc, q_b.and, q_b.or, q_b.neg, q_b.k)$$

Table 1: Spatio-Textual Objects

id	loc	txt
o1	(34.05, -118.24)	I go to Chipotle very often
o2	(31.95, -120.89)	Chipotle sauce is on discount
o3	(40.71, -74.01)	I enjoyed BBQ grill
o4	(37.77, -122.41)	Chipotle grill has really good taste
o5	(33.44, -112.07)	had a good time in BBQ grill
o6	(38.05, -120.16)	the Chipotle incident had huge impact

Where $q_b.loc$ is the query location, i.e., latitude and longitude of the query, $q_b.k$ is an integer, $q_b.and$ and $q_b.or$ are two sets of keywords, and $q_b.neg$ is a set of phrases.

Query q_b returns k objects o_i so that: (1) o_i is among spatial k nearest neighbors of $q_b.loc$, (2) $q_b.and \subset o_i.txt$, i.e., o_i includes all keywords $w_a \in q_b.and$, (3) $o_i.txt \bigcap q_b.or \neq \emptyset$, i.e., o_i includes at least one keyword $w_o \in q_b.or$, and (4) $q_b.negPh \nsubseteq o_i.txt \ \forall q_b.negPh \in q_b.neg$, i.e., o_i does not include any of the negative phrases.

Example. Suppose q_b is a BKQN instance with $q_b.loc = (34.25, -111.89)$, $q_b.and = \{grill\}$, $q_b.or = \{Chipotle, BBQ\}$, $q_b.neg = \{sauce\}$, $q_b.k = 1$. Objects $o3, o4, o5$ in Table 1 satisfy the textual predicates specified by q_b. Since $o5$ is the closest to q_b, the query returns $o5$ as the result.

4 TEQ INDEXING

This section introduces the Textual-Enhanced Quadtree (TEQ) index. Unlike all existing indexes, TEQ is distinguished in several aspects. First, TEQ is designed to effectively support negative keyword predicates. Such predicates are challenging as it introduces a counter-intuitive logic to the search process. In other words, the algorithm does not search for the presence of certain keywords, like in regular search, but the absence of these keywords, which is counter-intuitive. Second, TEQ is a parallelization-friendly index that enables parallel query processors to effectively consume its content. To this end, it uses a quadtree structure that ensures dynamic adaptation for cell content based on the data distribution to improve work-load balance in the parallel setting. Also, non-overlapping index cells ensure independence on processing cells in parallel. Third, TEQ efficiently supports both types of kNN spatial-keyword queries (top-k and Boolean queries) by storing both Boolean inclusion and weights of keywords in data objects. The rest of this section details the index structure and construction.

4.1 TEQ Index Structure

TEQ is a memory-resident hybrid index that combines the strength of quadtree for spatial partitioning and the strength of inverted index for efficient keyword organization. In specific, each quadtree leaf cell n of TEQ includes four indexing components: $n.ltp$, $n.neigh$, $n.iti$, and $n.obi$ that serve different purposes as follows:

(1) $n.ltp$ is the file name that acts as a location table pointer, which points to a hash file on disk that stores the mapping between $o.id$ and $o.loc$ for each object o in n. Figure 1 shows that the location of $o1$ is $(-76.1, 150.2)$. The location table is used to achieve $O(1)$ access for the locations of objects in leaf cell n.

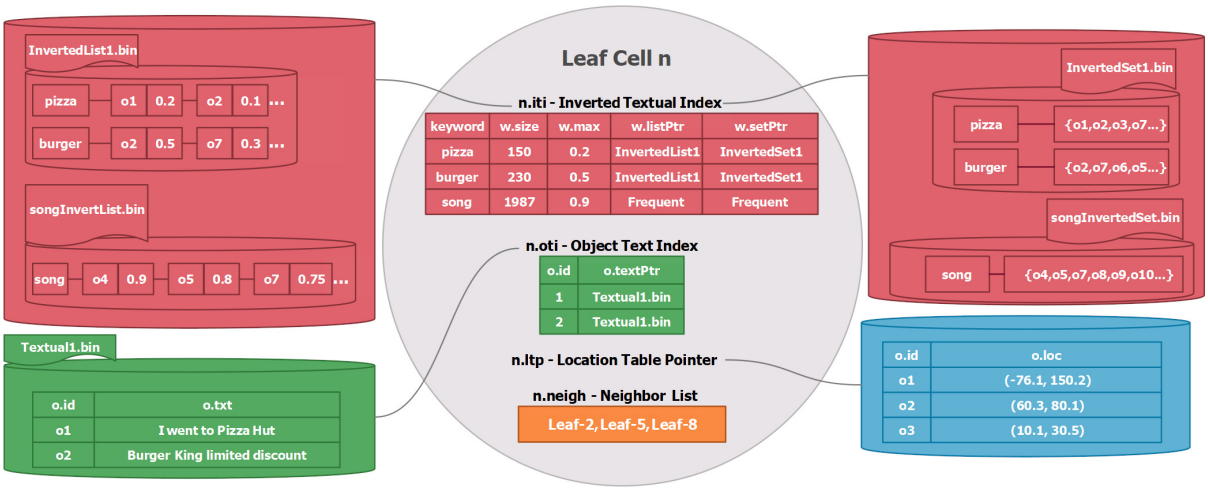


Figure 1: TEQ Leaf Cell Structure

(2) *n.neigh* is the list of spatially-neighboring cells of *n*. This list enables incrementally exploring a wider spatial range during query processing.

(3) *n.iti* is the inverted textual index that is realized as a hashtable whose key is a keyword *w* and the value is a quadruple $< w.size, w.max, w.listPtr, w.setPtr >$ interpreted as follows:

- $w.size$ is the number of objects that include w in the leaf cell n. For example, the word *pizza* appears 150 times (Figure 1).
- $w.max$ is the maximum possible weight of the keyword w among all the objects in cell n.
- $w.listPtr$ is the file name that acts as a pointer to the file that stores the inverted list of the keyword w. As shown in Figure 1, the inverted list stores $< o.id, s(o.w) >$ for each object o in cell n that includes w, where $s(o.w)$ is the weight of w as defined in Section 3. The list is sorted in descending order according to the weight. An inverted list is considered frequent if its file size is greater than the disk page size, e.g., 1MB, otherwise it is considered infrequent. Each frequent inverted list is stored in a separate file. Infrequent inverted lists are combined to make up files of the page size. The intuition is to minimize the number of I/Os to fetch the inverted lists corresponding to certain query keywords.
- $w.setPtr$ is the file name that points to the file that stores the set of object ids in cell n that includes w and the set is named inverted set. Different from inverted list, inverted set stores only the ids but not the weights, which serves a faster search under Boolean filtering conditions. Similar to lists, each inverted set is also categorized into either frequent or infrequent depending on the file size compared to the disk page size. Frequent sets are stored in separate files while infrequent sets are combined.

(4) *n.oti* is the object text index, realized as a hashtable that stores object id $o.id$ as a key and a pointer to the file storing the object full text $o.txt$. Each file that stores $o.txt$ is set to the page size and stores multiple textual descriptions corresponding to multiple objects. For example, in Figure 1, the textual description for $o1$ and $o2$ are stored in *Textual1.bin*.

4.2 TEQ Index Construction

Building TEQ indexing goes through two passes over the dataset. Constructing the index in two passes improves the indexing scalability, so even commodity machines with limited memory size could efficiently index large-sized datasets. The first pass builds the spatial indexing component. So, it inserts all the objects into the quadtree to shape its structure, builds the neighboring list *n.neigh* for each leaf cell *n*, and builds the location table pointer *n.ltp*. The second pass builds the textual indexing component. So, it builds the inverted textual index *n.iti* and the object text index *n.oti*. The rest of this section details each pass.

4.2.1 First Pass. Initially, all the objects are loaded from the disk and inserted into the quadtree in small-sized batches. In this pass, we store only $o.loc$ and $o.id$ for each object o in the main memory. Object locations are used in splitting quadtree cells when they have full capacity into four sibling leaf cells where each has a quarter of the parent's spatial area. Also, locations are used to redistribute objects over the new leaf cells after the split. The textual descriptions for the objects are ignored, due to the limited memory, to be processed in the second pass. Each leaf cell *n* dynamically maintains its neighboring leaf cells *n.neigh* during the splitting. A neighboring cell of *n* is any cell that shares a spatial border with *n* even if it is not a sibling cell. After inserting all objects, the location table for each leaf cell is constructed. For each leaf cell *n*, we traverse all the objects and insert them into a hashtable. Then, the hashtable is flushed to a file on disk and its pointer is maintained in *n.ltp*.

4.2.2 Second Pass. As the first pass places all the objects in leaf cells, the second pass builds the textual indexing components, *n.iti* and *n.oti*, for each leaf cell *n*. We read the objects of *n* from the disk. Then, two hashtables are initialized. The first hashtable is *n.idToinv* whose key is $o.id$ and the value is an empty inverted list. The second is *n.oti* that stores the mapping between $o.id$ and $o.txt$ as discussed

in Section 4.1. Then, for each object o, the textual description $o.txt$ is inserted into $n.oti$. Meanwhile, keywords of $o.txt$ are extracted using standard techniques. For each keyword w, the weight $s(o.w)$ is computed as the relative term frequency, which is the keyword count in $o.txt$ divided by the total number of keywords in $o.txt$. Other weighting functions can be used as well, as long as it is an invariant score, without impacting our proposed techniques. w is inserted into the corresponding inverted list in $n.idToinv$ along with its weight. After all the objects are processed, for each keyword in $n.idToinv$, we sort the corresponding inverted list based on the weight in descending order. We form the inverted set of objects from the inverted list by storing only object ids. The $w.max$ is set as the max weight in the inverted list, and $w.size$ is set as the number of entries in the list. The file names that point to the inverted list and the inverted set are stored as $w.listPtr$ and $w.setPtr$, respectively. Finally, the quadruple $< w.size, w.max, w.listPt, w.setPt >$ is then inserted into $n.iti$ as the value corresponding to keyword w. This process is repeated for each keyword and for each cell n until the whole index is constructed.

5 QUERY PROCESSING

In this section, we present *U-ASK* query processor to handle two types of kNN spatial-keyword queries; TKQN and BKQN queries as defined in Section 3. The query processor works in a parallelized master-worker paradigm that computes partial query results in worker nodes and then aggregates the final results in a master node. Our realization for a parallel node here is an application-level parallel thread. However, our distributed query processing framework could be easily extended to multi-machine distributed environments. The rest of this section introduces the high-level query processing framework, followed by details of processing both TKQN and BKQN queries.

Query processing framework. Our query processor works on a unified framework for both TKQN and BKQN queries. Both queries rank the answer objects based on some ranking function, either pure spatial ranking (BKQN) or spatio-textual ranking (TKQN). In addition, both queries filter out negative keyword predicates that are not supported in any existing work. So, given a query location $q.loc$, the query processor divides finding the top-k objects into multiple top-k sub-searches. While each sub-search is performed locally in an index leaf cell using a parallel worker node, a global top-k list is being aggregated by a master node to find the final answer.

The master node initially locates the index cell where the query location $q.loc$ lies. Then, the cell is inserted into a priority queue P with a priority distance zero. Then, a set of worker nodes are initiated to process cells in P in priority order. The master node dequeues an index cell n from P, and assigns it to one of the inactive worker nodes. Then, it inserts the neighboring cells of n ($n.neigh$) into P with priority score equals the spatial distance from the query location $q.loc$. The worker node finds a list of local top-k objects in n, L_n, and sends it back to the master node. This process repeats for each index cell in P. The master node keeps a global top-k list L. The local top-k lists L_n are merged into the global top-k list L. The process terminates when one of two conditions are satisfied. First, all index cells in P are processed. Second, the best cell in P cannot beat the current global top-k objects, i.e., the best priority score of any remaining object is worse than the k^{th} object in L. The best priority score of the remaining objects is computed as $q.\lambda \times S + (1 - q.\lambda)$ where S is the spatial score computed according to the minimum distance between $q.loc$ and the top cell in P.

Each worker node searches the top-k objects locally depending on the query type, either TKQN or BKQN, as detailed in the rest of the section. The global top-k list L is aggregated from local lists L_n in a straightforward way. L is implemented as a priority queue with the top-k ranking score as a priority score. For each local list L_n, the master node iterates over all objects. An object o is inserted in L with its priority score only if it is among the top-k scored objects maintained so far. Otherwise, o is ignored.

The rest of this section details the processing of TKQN and BKQN based on the described query processing framework.

5.1 TKQN Processing

This section presents the query processing algorithm POWER (Parallel bOttom-up search With incrEmental pRuning) to process TKQN query based on the described master-worker query processing framework.

Each worker node performs a local top-k search in an index cell n. First, we load n's the *Location Table* $n.LT$ into a memory buffer using $n.ltp$ if it is not already loaded. $n.LT$ stores the locations of all objects in n, which are frequently accessed during the query processing. Thus, buffering $n.LT$ in the main memory greatly reduces the I/O cost for retrieving the objects' locations. The memory buffer uses the Least Recently Used (LRU) policy for buffer management.

POWER is based on the TA algorithm [19] that performs a best-first search on sorted lists. TA algorithm retrieves the first entry in each sorted list in each iteration, and aggregates the score of each visiting entry by performing a random access to the other lists. The aggregated score is then used to update the top-k results. The algorithm keeps track of the upper bound score of unvisited entries by accumulating the first unvisited entry from each sorted list. If the upper bound score is less than the k^{th} score from the top-k results, then the remaining entries have no chance of being a top-k candidate and the searching process terminates.

TKQN query ranks its results based on two attributes, spatial and textual attributes. TEQ index already stores keyword inverted lists sorted by the textual weight, which can be accessed by $n.iti$. However, the spatial ordering depends on the query location $q.loc$. So, POWER sorts all objects within a cell n on the fly based on the spatial distance from the query location $q.loc$. The objects are inserted into a priority queue based on a normalized spatial score, in the range $[0, 1]$ as a priority score. The normalized spatial score has the same range as the textual weight. This priority queue facilitates incremental retrieval of the closest objects with small overhead. Then, POWER works on a set of sorted lists, one spatially-ordered list and a set of textually-ordered lists, each list corresponds to one keyword in $q.pos$. While retrieving objects in ascending order of score based on the TA algorithm, the keyword predicates are evaluated on each retrieved object o. In TKQN, keyword predicates include both positive keyword predicates, represented by $q.pos$, and negative keyword predicates, represented by $q.neg$. For the positive keyword predicates, POWER checks if the textual description $o.txt$ includes

any of the keywords in the set $q.pos$. For the negative keyword predicates, POWER checks if $q.negPh \subset o.txt \; \forall q.negPh \in q.neg$. Negative phrases, $q.negPh$, are checked in ascending order of length, i.e., the number of words in the phrase. The reason is that shorter phrases have a higher probability of occurrence. For a specific negative phrase $q.negPh$, POWER first checks if $o.txt$ includes all the words in $q.negPh$ by looking up the inverted sets corresponding to the different keywords in $q.negPh$ using $n.iti$. This works as a quick and efficient filter to eliminate irrelevant candidates early. If at least one keyword from $q.negPh$ is not included in $o.txt$, then it is clear that $o.txt$ does not include $q.negPh$. Otherwise, further verification is needed. In this case, an I/O request is sent to retrieve $o.txt$ by using $n.oti$ to further examine whether the whole $q.negPh \subset o.txt$. If o includes any of the negative phrases in $q.neg$, it will not be considered a valid candidate.

Despite the optimizations of POWER algorithm, it is still slow because it builds a list (priority queue) of distances for all objects in each index cell on the fly, which is computationally expensive. Such list cannot be pre-computed because it depends on the query location, which varies with every query. To alleviate such high cost, we further extend POWER with textual and spatial pruning strategies that significantly reduce the query processing cost, yet, still ensure exact query results. The rest of the section proposes two variants of POWER, POWER with Textual pruning (POWER-T) and POWER with Spatial pruning (POWER-S).

5.1.1 ***POWER-T.*** POWER-T focus on textual pruning, while skipping the difference in the spatial score in the early search since the objects from the same leaf cell are close in space, thus have close spatial scores. The general idea of POWER-T is to first identify k candidate objects based on TA algorithm [19] applied to the textual inverted lists within an index cell n. The k^{th} score is marked as the lower bound $score_k$. Then POWER-T sequentially visits the remaining objects from the textual inverted lists using TA algorithm with spatial score computed using the $n.spatialUpperScore$, i.e., the spatial upper bound score of the cell, until in some iteration, the score of an object is less than $score_k$. Then the remaining objects can be safely pruned without further exploration. The intuition is, the objects in the same leaf cell n are close in space and have a close spatial score. Thus, the textual score plays a more important role in determining the top-k. POWER-T is detailed as follows.

Initially, POWER-T performs the TA algorithm on the textual inverted lists corresponding to the words in $q.pos$ to find the initial k feasible objects. In each iteration, the first object o that has the greatest score is retrieved from the textual inverted list. Then POWER-T evaluates the keyword predicates, both positive and negative keyword predicates, in the same way described in the original POWER algorithm. If o does not violate any negative keyword predicates, $score(o, q)$ is computed (based on Equation 1), through aggregating o's spatial and textual scores from the location table and different textual inverted lists, respectively. To speed up aggregation from textual inverted lists, we build a corresponding hashtable on the fly for each inverted list, with object id as a key and textual score as a value. Note that the initial k feasible objects we identify have promising textual scores, since they are incrementally retrieved from the textual inverted lists with textual scores sorted in descending order. The spatial scores of these objects, however, have no guarantees. Even though, the spatial score plays a less important role because the objects within the same leaf index cell are spatially close and thus have close spatial score.

After the initial k feasible objects are identified, POWER-T continues to perform TA algorithm on the sorted textual inverted lists. In each iteration, POWER-T retrieves the best object o following TA algorithm and places o in a list l. POWER-T keeps track of and updates the textual upper bound score t_{upper}, which is computed to be the sum of the first unvisited entry from each textual inverted list. t_{upper} monotonically decreases in each iteration because the objects in the inverted textual lists with high scores are being processed incrementally. The spatial upper bound score, denoted as $n.spatialUpperBound$, is computed as the score corresponding to the minimum distance between $q.loc$ and the cell n. At some iteration, when $q.\lambda \times n.spatialUpperBound + (1 - q.\lambda) \times t_{upper} \leq score_k$, the search could be terminated and the remaining unvisited objects could be safely eliminated because their upper bound score is less than the pruning score $score_k$. POWER-T computes $score(o, q)$ for all the objects in the list l, which are the objects that cannot be pruned and need further examination, and updates the top-k results accordingly.

5.1.2 ***POWER-S.*** POWER-S uses a similar approach to POWER-T but differs in that it focuses on applying spatial pruning instead of textual pruning. The general idea of POWER-S is to visit the objects incrementally by their distance to $q.loc$, while ignoring the textual scores and using the maximum textual score instead, until the terminating condition is satisfied.

Similar to POWER, retrieving the objects incrementally based on the distance to $q.loc$ puts a high computational cost on POWER-S because $q.loc$ varies for each query q. So, the distances to $q.loc$ are computed and sorted on the fly and cannot be precomputed as the textual scores. POWER-S accelerates this procedure by dividing each leaf cell into smaller equal-sized blocks with each block including a portion of the objects during the indexing. POWER-S retrieves the blocks of the leaf cell and places all the blocks into a priority queue with priority score equals to the minimum distance to $q.loc$.

Similar to POWER-T, POWER-S first retrieves the initial k feasible objects and computes a lower bound score. POWER-S retrieves an entry from the priority queue in each iteration. If the entry is a block, then we put all the objects within the block into the priority queue with priority score equals to the distance between each object to $q.loc$. Otherwise, the entry is an object o and POWER-S evaluates the keyword predicates on $o.txt$ in the same way described before. POWER-S repeats above in each iteration until the initial k feasible objects are identified. Then POWER-S denotes the pruning score as the minimum score $score_k$.

After the initial k feasible objects are identified, POWER-S keeps retrieving entries in each iteration. If the entry is an object o that satisfies the keyword predicates, then we place o in a list l. POWER-S keeps track of the spatial upper bound score s_{upper}, which is the spatial score of the first unvisited object in the priority queue. The textual upper bound score, denoted as $n.textualUpperBound$, is computed as the sum of the first object in each textual inverted list, which can be accessed by $w.max$ for each keyword w through $n.iti$. $n.textualUpperBound$

is truncated to 1 if it is greater than 1. At some iteration, when $q.\lambda \times s_{upper} + (1-q.\lambda) \times n.textualUpperBound \leq score_k$, the search is terminated and the remaining unvisted objects could be safely eliminated because their upper bound score is less than the pruning score $score_k$. Then POWER-S computes $score(o, q)$ for all the objects in l, which are the objects that cannot be pruned and need further examination, and updates the top-k results accordingly.

For large $q.\lambda$ values, the spatial score is more important in Equation 1 and vice versa. Ideally, a good approach is to set a threshold in the query processor so that when λ is greater than the threshold, POWER-S is invoked and when λ is smaller than the threshold, then POWER-T is invoked to adapt the appropriate pruning criteria. We experimentally explore the best threshold value in Section 6.2.1.

5.2 BKQN Processing

This section proposes POWER-BF query processing algorithm to process BKQN query as defined in Section 3. POWER-BF still works on the same master-worker paradigm that is introduced for POWER, POWER-T, and POWER-S. However, the original POWER algorithm and its variants cannot be applied as is for BKQN query as it does not use textual score in ranking as in TKQN query. In addition, BKQN has both AND and OR conjunctions in their keyword predicates, which is not the case in TKQN.

The worker node in POWER-BF initializes a priority queue with priority score equals to the spatial distance to the query location $q.loc$ to store all the objects that satisfy the textual predicates. The worker first considers the *AND* predicate and sorts the *AND* keywords $w \in q.and$ based on $w.size$ in ascending order. Then, the worker starts from the *AND* keyword w with minimum $w.size$, and takes the intersection and stores the results in a set s_{and}. This is because intersecting in ascending order results in the minimum number of comparisons. After all the *AND* keywords are considered, if s_{and} is not empty, then the worker starts to process the *OR* words from $q.or$. The worker creates a new set s_{or}, to take the union of the set for each $w \in q.or$, regardless of the order. Then, the worker takes the intersect of s_{and} and s_{or} to produce a new set s. For each object $o \in s$, o satisfies the $q.and$ and $q.or$ constraints. After that, the worker evaluates negative keyword predicates in the same way described for other POWER variants. If o satisfies the negative keyword predicates, it is inserted in the priority queue. Finally, the worker node returns the top-k objects from the priority queue to the master node as the local top-k results.

6 EXPERIMENTAL EVALUATION

This section presents extensive experimental evaluation of *U-ASK* framework including its TEQ indexing and the different variants of POWER algorithm. Section 6.1 presents the experimental setup. Sections 6.2 present the performance evaluation for the different parameters and framework components. Section 6.3 and Section 6.4 compare the proposed algorithms against the state-of-the-art under TKQN and BKQN, respectively.

6.1 Experimental Setup

We evaluate *U-ASK*'s TEQ indexing and the query processing under different queries, i.e., TKQN with POWER, POWER-T, and POWER-S, and BKQN with POWER-BF. Our algorithms are compared against the state-of-the-art for both TKQN and BKQN queries. The state-of-the-art algorithm that solves the closest problem to TKQN is RCA [42]. RCA uses a best-first search paradigm to support the top-k kNN spatial-keyword query and index locations using Z-order curves. We modify RCA to support TKQN problem. When an object is considered as a candidate in the top-k objects by RCA, the algorithm evaluates the negative keyword predicates using the same approach discussed for POWER.

The state-of-the-art algorithm that solves the closest problem to BKQN is SKI [5]. SKI handles the Boolean kNN that supports a conjunction of *AND*, *OR*, and negative keyword predicates by performing a best-first search on an R-Tree [22] of objects with bitmap stored in the intermediate nodes. It uses negative keyword predicates that are bags-of-words, so the sequence of the phrase is ignored. We modify SKI to support BKQN queries as follows. The negative keyword predicates are evaluated in R-tree leaf nodes by reading the textual description of all objects and checking for negative phrases. This is because the bitmap can be used for individual words but not phrases.

Table 2: Evaluation Parameters Values

Parameter	Values
Dataset Size (million)	2, **4**, 6, 8, 10
Number of Threads	1, 2, **4**, 8
Buffer Size (MB)	50, 100, 150, **200**, 250, 300
$\|q.pos\|$	1 ,2 ,**3**, 4, 5
$\|q.neg\|$	**1**, 2, 3, 4, 5
$q.negLen$	1, **2**, 3, 4, 5
$q.\lambda$	0, 0.1, 0.3, **0.5**, 0.7 ,0.9 , 1
$q.k$ (TKQN)	5, **10**, 50, 100, 500, 1000, 5000, 10000
$q.k$ (BKQN)	**10**, 30, 50, 70, 90
$\|q.and\|$	1, **2**, 3, 4
$\|q.or\|$	1, **2**, 3, 4

Evaluation dataset. Our evaluation uses 10 millions tweets collected from Twitter APIs [37]. We compose subsets of sizes ranging from 2 millions to 10 millions tweets. Each tweet includes an id, a location, i.e., latitude and longitude, and a textual description, which matches the definition of spatio-textual objects in Section 3.

Query workload and parameters. Out of real tweets, we generate queries as follows. We extract the top frequent phrases with different lengths, e.g., burger, pizza hut, chipotle mexican grill. For a TKQN query q_t, the $q_t.loc$ is generated as a random point in the space, $q_t.pos$ is randomly selected from single-word phrases, $q_t.neg$ is randomly selected from phrases of corresponding length ($q_t.negLen$), and $q_t.\lambda$ is randomly chosen from a set of values that span the whole range $[0, 1]$. For a BKQN query q_b, $q_b.loc$ and $q_b.neg$ are generated in the same way as TKQN query workload. $q_b.or$ and $q_b.and$ words are randomly selected from phrases of various sizes, depending on the sizes $|q_b.or|$ and $|q_b.and|$. The parameter values of our evaluation is shown in Table 2, the default values are emphasized in boldface.

6.2 Parameter Tuning

Appendix A evaluates parameters of TEQ index due to its stability for different parameters. The rest of this section evaluates the query processing parameters.

6.2.1 λ ***value***. Figure 2 shows different values for a threshold on λ value in TKQN query. If λ is less than the threshold, the query processor uses POWER-T, otherwise, it uses POWER-S. The intuition is that small values of λ give higher importance for textual scores, which gives an advantage for textual pruning of POWER-T. However, Figure 2 shows that the optimal query latency is achieved when the threshold is 1. This means in all cases, POWER-T outperforms POWER-S, even with large λ values. The reason for such behaviour is twofold. First, POWER-S incrementally visits the objects based on the distance to the query but gives no guarantee on whether each visited object includes at least one query positive keywords, whereas each object visited by POWER-T includes at least one query keywords so it produces more relevant candidate objects faster. Second, TEQ organizes the objects in a quadtree structure that adapts to the spatial distribution of the data. This means the objects within the same leaf cell have relatively close spatial proximity and thus have close spatial scores with regard to a specific query due to how quadtree organizes objects. On the other hand, different objects within the same leaf node might have significantly different textual scores. The greater difference in textual scores and less difference in spatial scores lead to the fact that pruning based on textual dimension is more efficient and prunes more objects.

6.2.2 ***Buffer size***. The buffer size is the maximum memory used to buffer the *Location Table* of index leaf cells. Figure 3 shows that the query latency decreases when the buffer size increases as expected. In addition, small buffer size of 200 MB is enough to achieve a great trade-off between memory usage and query latency of a few milli-seconds. This shows the scalability and affordability of POWER algorithms even on commodity hardware.

6.2.3 ***Number of Threads***. Figure 4 shows the effect of the number of threads. For small k, i.e., k < 100, multi-threading does not demonstrate a clear advantage. This is because when k is small, the bottleneck is in I/O, which cannot be accelerated with more computing cores. When k is large, heavy computation is required and the bottleneck is CPU, which is accelerated using more computing cores. We apply single-threaded version of our algorithms for small k, i.e., k < 100, and multi-threaded version for large k.

6.3 TKQN Query Evaluation

Section 6.2 shows that POWER-T outperforms POWER and POWER-S in all cases. The key strength of POWER-T is generating more relevant candidate objects from textual inverted lists and naturally prunes irrelevant objects that do not include any query keyword. This section compares POWER-T against RCA [42] to solve TKQN queries under different settings.

6.3.1 ***The effect of query keywords***. Figure 5a shows the query latency of POWER-T compared to RCA under different number of positive query keywords $q.pos$. The runtime of RCA increases as the number of positive keywords increases whereas the runtime of POWER-T remains stable at around 5 ms for different $|q.pos|$ values. POWER-T significantly outperforms RCA with up to 3.8× speedup thanks to its effective pruning techniques, efficient object retrieval through TEQ index structure, and main-memory buffering of spatial locations. On the other hand, RCA is slower than POWER-T because visiting objects either with high spatial scores or high textual scores enlarges the search scope. The method lacks the mechanism to efficiently identify the objects that have both promising spatial and textual scores at the same time, which degrades its performance.

The experiment shows the query latency of POWER-T compared to RCA under different number of negative phrases $|q.neg|$ (Figure 5b) and different number of keywords in each negative phrase $q.negLen$ (Figure 5c). The query latency of both algorithms increases as $|q.neg|$ or $q.negLen$ increases. This is because the increasing of $|q.neg|$ or $q.negLen$ incurs more I/Os to load the inverted sets corresponding to the negative keywords. At the same time, it also results in longer latency when evaluating negative keyword predicates. In all cases, POWER-T significantly outperforms RCA and it is up to 5.25× faster. The reason behind the good performance is similar to the one explained above.

6.3.2 ***The effect of weighting factor.*** With larger λ, the spatial score becomes more important than the textual score in Equation 1 and vice versa. When λ is 0 or 1, the TKQN query falls into the extreme cases as the ranking purely depends on either spatial proximity or textual relevance. Figure 5d shows that POWER-T outperforms RCA in all cases with up to 19× speedup. Even in extreme cases, POWER-T still has great performance due to its efficiency and effectiveness in locating objects with promising scores.

6.3.3 ***The effect of k.*** Figure 5e shows the effect of $q.k$, i.e., the answer size. POWER-T is up to 13.5× faster than RCA. The runtime of both POWER-T and RCA increases when k increases. POWER-T performs significantly better than RCA under larger k because larger k results in heavier computation on CPU, which is perfectly handled in the multi-threading master-worker architecture in POWER-T.

6.3.4 ***The effect of dataset size***. Figure 5f shows the effect of different dataset sizes. POWER-T still outperforms RCA in all cases and it is up to 3.4× faster. With increasing dataset sizes, the query latency of POWER-T grows significantly slower than RCA, which shows the great scalability of POWER-T to handle large-sized datasets.

6.4 BKQN Query Evaluation

This section evaluates the performance of POWER-BF to solve BKQN queries. We compare POWER-BF against SKI [5] under different parameter settings. As Table 2 shows, the k value in BKQN is set to smaller values compared to TKQN. This is because BKQN places more requirement on the textual predicates and the corresponding resulting set is smaller.

6.4.1 ***The effect of AND keywords***. Figure 6a shows the effect of $|q.and|$, i.e., the size of *AND* keywords set. POWER-BF significantly outperforms SKI in all cases and achieves up to 34.8× better latency compared to SKI. The reason behind the good performance is that POWER-BF is able to precisely locate the objects that have promising scores by utilizing the TEQ index. With increasing number of *AND* keywords, the latency of POWER-BF remains stable

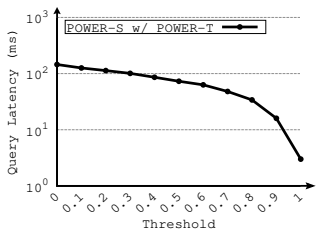


Figure 2: The Effect of λ Threshold

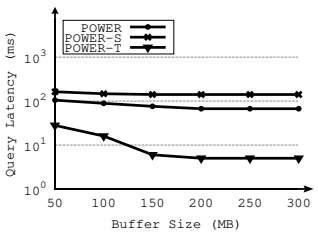


Figure 3: The Effect of Buffer Size

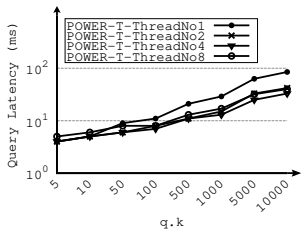
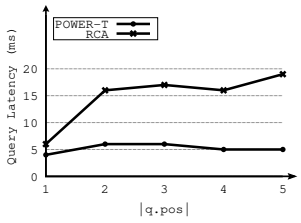
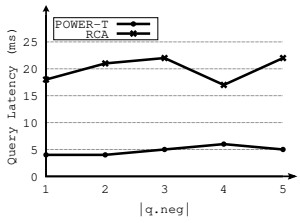


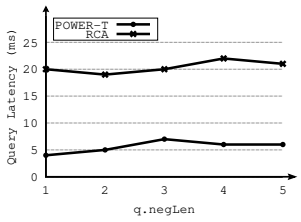
Figure 4: The Effect of Number of Threads



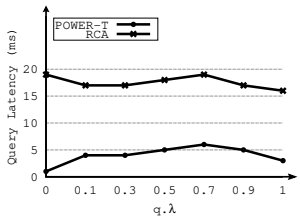
(a) Number of positive words



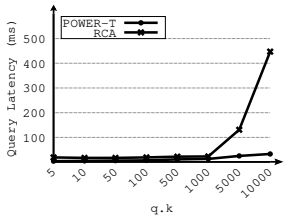
(b) Number of negative phrases



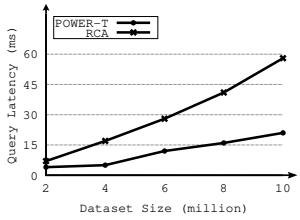
(c) Length of negative phrases



(d) Weighting factor λ

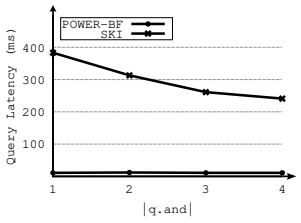


(e) k

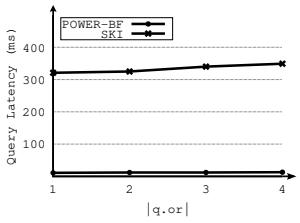


(f) Dataset Size

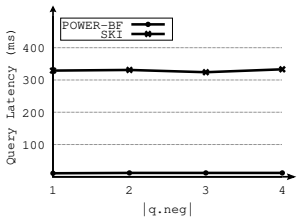
Figure 5: TKQN Query Evaluation



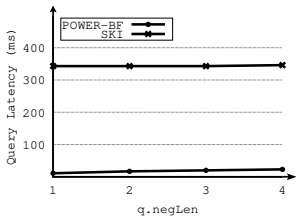
(a) Number of AND words



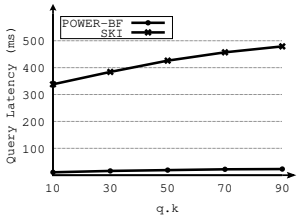
(b) Number of OR words



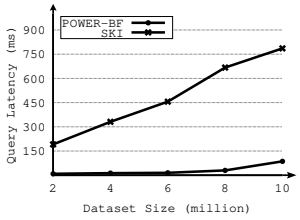
(c) Number of negative phrases



(d) Length of negative phrases



(e) k



(f) Dataset Size

Figure 6: BKQN Query Evaluation

while SKI latency decreases. This is because with more keywords, more sub-trees in SKI could be eliminated during the query processing without further exploration by looking up the bitmap, which reduces the query processing time.

6.4.2 ***The effect of OR keywords.*** Figure 6b shows the effect of $|q.or|$, i.e., the size of *OR* keywords set. POWER-BF still significantly outperforms SKI in all cases with up to 29.1× speedup for utilizing the TEQ index effectively. The latency of SKI increases when the number of *OR* keywords increases because more objects become candidates, while POWER-BF latency is almost stable.

6.4.3 ***The effect of negative phrases.*** Figure 6c and Figure 6d show that POWER-BF outperforms SKI under different number of negative phrases and different number of keywords in each negative phrase, respectively. POWER-BF is still much faster than SKI, up to 31.1×. Both the runtime of POWER-BF and SKI remain stable as the size of $q.neg$ changes.

6.4.4 ***The effect of k.*** Figure 6e shows the effect of the answer size $q.k$. Latency of both POWER-BF and SKI increases as $q.k$ increases because more computation is involved. However, POWER-BF is still significantly faster with up to 30.7× compared to SKI.

6.4.5 ***The effect of dataset size.*** Figure 6f shows that increasing the dataset size increases latency of both POWER-BF and BKQN. POWER-BF achieves up to 30.4× faster than SKI. The latency of POWER-BF grows much slower than SKI, which demonstrates the significant scalability of POWER-BF on large-sized datasets.

7 CONCLUSION

This paper proposes a unified framework *U-ASK* to support kNN spatial-keyword queries, both top-k and Boolean queries, with negative keyword predicates. We define TKQN and BKQN queries that extend traditional queries with negative keyword predicates to maximize the expressiveness of spatial keyword queries. To support the new queries, we propose a hybrid TEQ index and several query processing algorithms that employ a master-worker paradigm to exploit the index content and provide highly efficient query latency. Our experimental evaluation on real datasets has shown superior performance for all our algorithms with an order of magnitude faster runtime compared to the state-of-the-art algorithms. Besides the superior performance, our proposed framework supports various queries types simultaneously using a unified underlying architecture, which is favorable for system administrators.

REFERENCES

[1] J. Benhardus and J. Kalita. Streaming trend detection in twitter. *International Journal of Web Based Communities*, 9(1):122–139, 2013.

[2] U. C. Bureau. Food Poisoning at Chipotle, 2021. https://abcnews.go.com/US/wireStory/chipotle-agrees-record-25-million-fine-tainted-food-70271636.

[3] X. Cao, G. Cong, T. Guo, C. S. Jensen, and B. C. Ooi. Efficient processing of spatial group keyword queries. *ACM Transactions on Database Systems (TODS)*, 40(2):1–48, 2015.

[4] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 373–384, 2011.

[5] A. Cary, O. Wolfson, and N. Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In *International Conference on Scientific and Statistical Database Management*, pages 87–95, 2010.

[6] H. K.-H. Chan, C. Long, and R. C.-W. Wong. Inherent-cost aware collective spatial keyword queries. In *International Symposium on Spatial and Temporal Databases*, pages 357–375, 2017.

[7] J. Chen and W. Jiang. Context-aware personalized POI sequence recommendation. In *International Conference on Smart City and Informatization*, pages 197–210, 2019.

[8] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *Proceedings of the VLDB Endowment*, 6(3):217–228, 2013.

[9] L. Chen, X. Lin, H. Hu, C. S. Jensen, and J. Xu. Answering why-not questions on spatial keyword top-k queries. In *2015 IEEE 31st International Conference on Data Engineering*, pages 279–290. IEEE, 2015.

[10] L. Chen, S. Shang, C. Yang, and J. Li. Spatial keyword search: a survey. *GeoInformatica*, 24(1):85–106, 2020.

[11] L. Chen, J. Xu, X. Lin, C. S. Jensen, and H. Hu. Answering why-not spatial keyword top-k queries via keyword adaption. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 697–708. IEEE, 2016.

[12] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 277–288, 2006.

[13] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: efficient geo-search query processing. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 423–432, 2011.

[14] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *Proceedings of the VLDB Endowment*, 2(1):337–348, 2009.

[15] N. Cui, J. Li, X. Yang, B. Wang, M. Reynolds, and Y. Xiang. When geo-text meets security: privacy-preserving boolean spatial keyword queries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1046–1057, 2019.

[16] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *2008 IEEE 24th International Conference on Data Engineering*, pages 656–665, 2008.

[17] Deng, Ke and Li, Xin and Lu, Jiaheng and Zhou, Xiaofang. Best keyword cover search. *IEEE Transactions on Knowledge and Data Engineering*, 27(1):61–73, 2014.

[18] K. Domdouzis, B. Akhgar, S. Andrews, H. Gibson, and L. Hirsch. A social media and crowdsourcing data mining system for crime prevention during and post-crisis situations. *Journal of Systems and Information Technology*, 2016.

[19] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[20] Y. Gao, J. Zhao, B. Zheng, and G. Chen. Efficient collective spatial keyword query processing on road networks. *IEEE Transactions on Intelligent Transportation Systems*, 17(2):469–480, 2015.

[21] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, 1982.

[22] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[23] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *19th International Conference on Scientific and Statistical Database Management (SSDBM 2007)*, pages 16–16, 2007.

[24] H.-J. Hong, G.-M. Chiu, and W.-Y. Tsai. A single quadtree-based algorithm for top-k spatial keyword query. *Pervasive and Mobile Computing*, 42:93–107, 2017.

[25] W. Huang, G. Li, K.-L. Tan, and J. Feng. Efficient safe-region construction for moving top-k spatial keyword queries. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 932–941, 2012.

[26] J. Li, H. Wang, J. Li, and H. Gao. Skyline for geo-textual data. *GeoInformatica*, 20(3):453–469, 2016.

[27] Y. Li, J. Huang, M. Fan, J. Lei, H. Wang, and E. Chen. Personalized query auto-completion for large-scale POI search at Baidu Maps. *ACM Transactions on Asian and Low-Resource Language Information Processing (TALLIP)*, 19(5):1–16, 2020.

[28] Z. Li, K. C. Lee, B. Zheng, W.-C. Lee, D. Lee, and X. Wang. Ir-tree: An efficient index for geographic document search. *IEEE transactions on knowledge and data engineering*, 23(4):585–599, 2010.

[29] Y. Ma, Y. Zhang, and X. Meng. St-hbase: a scalable data management system for massive geo-tagged objects. In *International Conference on Web-Age Information Management*, pages 155–166, 2013.

[30] A. Madani, O. Boussaid, and D. E. Zegour. Real-time trending topics detection and description from Twitter content. *Social Network Analysis and Mining*, 5(1):1–13, 2015.

[31] A. Magdy, A. M. Aly, M. F. Mokbel, S. Elnikety, Y. He, S. Nath, and W. G. Aref. GeoTrend: spatial trending queries on real-time microblogs. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–10, 2016.

[32] A. R. Mahmood, W. G. Aref, A. M. Aly, and M. Tang. Atlas: on the expression of spatial-keyword group queries using extended relational constructs. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–10, 2016.

[33] A. Sadilek, H. Kautz, L. DiPrete, B. Labus, E. Portman, J. Teitel, and V. Silenzio. Deploying nEmesis: Preventing foodborne illness by data mining social media. In *Twenty-Eighth IAAI Conference*, 2016.

[34] M. S. C. Sapul, T. H. Aung, and R. Jiamthapthaksin. Trending topic discovery of Twitter Tweets using clustering and topic modeling algorithms. In *2017 14th international joint conference on computer science and software engineering (JCSSE)*, pages 1–6, 2017.

[35] J. Shi, D. Wu, and N. Mamoulis. Textually relevant spatial skylines. *IEEE Transactions on Knowledge and Data Engineering*, 28(1):224–237, 2015.

[36] Y. Tao and C. Sheng. Fast nearest neighbor search with keywords. *IEEE transactions on knowledge and data engineering*, 26(4):878–888, 2013.

[37] Twitter. Twitter Developer API, 2022. https://developer.twitter.com/en/docs/twitter-api.

[38] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In *International Symposium on Spatial and Temporal Databases*, pages 218–235, 2005.

[39] D. Wu, B. Choi, J. Xu, and C. S. Jensen. Authentication of moving top-k spatial keyword queries. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):922–935, 2014.

[40] D. Wu, M. L. Yiu, and C. S. Jensen. Moving spatial keyword queries: Formulation, methods, and analysis. *ACM Transactions on Database Systems (TODS)*, 38(1):1–47, 2013.

[41] C. Zhang, Y. Zhang, W. Zhang, and X. Lin. Inverted linear quadtree: Efficient top k spatial keyword search. *IEEE Transactions on Knowledge and Data Engineering*, 28(7):1706–1721, 2016.

[42] D. Zhang, C.-Y. Chan, and K.-L. Tan. Processing spatial keyword query as a top-k aggregation query. In *Proceedings of the 37th international ACM SIGIR conference on research & development in information retrieval*, pages 355–364, 2014.

[43] D. Zhang, K.-L. Tan, and A. K. Tung. Scalable top-k spatial keyword search. In *Proceedings of the 16th international conference on extending database technology*, pages 359–370, 2013.

[44] K. Zhao, G. Cong, Q. Yuan, and K. Q. Zhu. SAR: A sentiment-aspect-region model for user preference analysis in geo-tagged reviews. In *2015 IEEE 31st international conference on data engineering*, pages 675–686, 2015.

[45] S. Zhao and L. Xiong. Group nearest compact POI set queries in road networks. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*, pages 106–111, 2019.

[46] K. Zheng, H. Su, B. Zheng, S. Shang, J. Xu, J. Liu, and X. Zhou. Interactive top-k spatial keyword queries. In *2015 IEEE 31st International Conference on Data Engineering*, pages 423–434, 2015.

[47] W. Zhou and W. Han. Personalized recommendation via user preference matching. *Information Processing & Management*, 56(3):955–968, 2019.

Table 3: Index Parameters Values

Parameter	Values
TEQ Cell Size	10k, 20k, **30k**, 40k, 50k
TEQ Depth	5, 10, **15**, 20, 25
TEQ Spatial Blocks	25, 100, **225**, 400, 625

APPENDIX

A INDEX PARAMETER TUNING

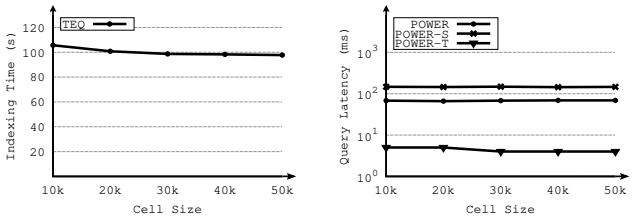
This section evaluates different index parameters.

TEQ quadtree shape. The shape of the TEQ quadtree is determined by the cell size and the maximum depth of the tree. The cell size in TEQ index is the maximum number of objects within each leaf index cell. Figure 7 illustrates how the TEQ cell size affects the indexing time and TKQN query latency, respectively. The result shows that both indexing time and query latency are slightly affected by the TEQ cell size. Having small cell sizes leads to more number of leaf cells but each leaf cell takes less storage, which achieves a trade-off compared to large-sized leaf cells. With regards to query processing, for a specific TKQN query under smaller cell size using either POWER, POWER-T or POWER-S, it takes less time to find the local top-k objects from a specific leaf cell but usually requires exploring more cells.

In all cases, POWER-T outperforms POWER and POWER-S. The high cost of POWER comes from building the priority queue based on distance to *q.loc*, which requires heavy computation compared to retrieving textual inverted lists. The high cost of POWER-S comes from retrieving the object incrementally by the distance to *q.loc*, which does not guarantee that the object includes at least one query positive keyword. In fact, the objects including at least one of the query keyword only take a small portion of all the objects. Thus, POWER-S ends up visiting lots of objects that do not satisfy the textual predicates. POWER-T is efficient because it prunes the search space through textual dimension and the objects retrieved by POWER-T include at least one query positive keywords.

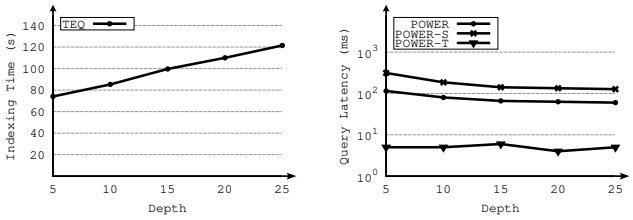
The depth of TEQ index refers to the maximum depth that the quadtree can reach. Figure 8 illustrates how the TEQ depth limit affects the indexing time and TKQN query processing time using POWER, POWER-S, and POWER-S. As the depth increases, the indexing time increases and the query latency slightly decreases. In fact, the decrease in query latency is much less than the increase in indexing time. POWER-T still outperforms POWER and POWER-S in all cases.

TEQ spatial blocks number. The TEQ spatial blocks number, detailed in Section 5.1.2, refers to the number of spatial blocks further divided from each leaf cell, which is proposed to speed up the process of incrementally retrieving the nearest neighbor objects to *q.loc* in POWER-S. Figure 9 illustrates the effect of the number of spatial blocks in each leaf cell on the indexing time and query processing time. The indexing time is slightly affected by the number of spatial blocks. This is because the total number of mappings does not change, regardless of the number of blocks. More blocks lead to less number of mappings in each block and vice versa. The query processing time for POWER-S slightly decreases as the number of blocks increases. This is because the increasing number of blocks results in fewer entries in the priority queue during the POWER-S query processing, which reduces the processing time. The query processing time for POWER and POWER-T is not affected by the number of spatial blocks.



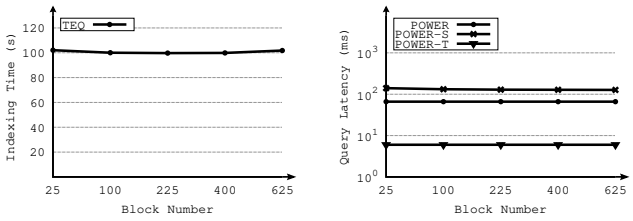
(a) Indexing Time **(b) Query Latency**

Figure 7: The Effect of TEQ Cell Size



(a) Indexing Time **(b) Query Latency**

Figure 8: The Effect of TEQ Depth



(a) Indexing Time **(b) Query Latency**

Figure 9: The Effect of TEQ spatial blocks number