The Flexible Group Spatial Keyword Query

Sabbir Ahmad
Dept of Computer Science & Eng
Bangladesh Univ of Eng & Tech
Dhaka, Bangladesh
ahmadsabbir@cse.buet.ac.bd

Rafi Kamal
Dept of Computer Science & Eng
Bangladesh Univ of Eng & Tech
Dhaka, Bangladesh
rafikamalb@gmail.com

Mohammed Eunus Ali
Dept of Computer Science & Eng
Bangladesh Univ of Eng & Tech
Dhaka, Bangladesh
eunus@cse.buet.ac.bd

Jianzhong Qi
University of Melbourne
Melbourne, Australia
jianzhong.qi@unimelb.edu.au

Peter Scheuermann
Northwestern University
Illinois, USA
peters@eecs.northwestern.edu

Egemen Tanin
University of Melbourne
Melbourne, Australia
etanin@unimelb.edu.au

ABSTRACT

We present a new class of service for location based social networks, called the Flexible Group Spatial Keyword Query, which enables a group of users to collectively find a point of interest (POI) that optimizes an aggregate cost function combining both spatial distances and keyword similarities. In addition, our query service allows users to consider the trade-offs between obtaining a suboptimal solution for the entire group and obtaining an optimimized solution but only for a subgroup.

We propose algorithms to process three variants of the query: (i) the group nearest neighbor with keywords query, which finds a POI that optimizes the aggregate cost function for the whole group of size n, (ii) the subgroup nearest neighbor with keywords query, which finds the optimal subgroup and a POI that optimizes the aggregate cost function for a given subgroup size m ($m \leq n$), and (iii) the multiple subgroup nearest neighbor with keywords query, which finds optimal subgroups and corresponding POIs for each of the subgroup sizes in the range $[m, n]$. We design query processing algorithms based on branch-and-bound and best-first paradigms. Finally, we provide theoretical bounds and conduct extensive experiments with two real datasets which verify the effectiveness and efficiency of the proposed algorithms.

1 INTRODUCTION

The *group nearest neighbor* (GNN) query [19] and its variants, the flexible aggregate nearest neighbor (FANN) [15] query and the consensus query [1] have been previously studied in the spatial database domain. Given a set Q of n queries and a dataset D, a GNN query finds the data object that minimizes the aggregate distance (e.g., sum or max) for the group, whereas an FANN query finds the optimal subgroup of query points and the data object that minimizes the aggregate distance for a subgroup of size m, and a consensus query finds optimal subgroups and the data objects for each of the subgroup sizes in the range $[n', n]$ for $n' < n$. In all these studies, the aggregate similarity is computed based on only spatial (or Euclidean) distances between a data point and a group of query points. In this paper, we address variants of the above queries in the context of the *spatial textual* domain, where both spatial proximity and keyword similarity for a *group or subgroups of users* to data points need to be considered. We call this class of query the *flexible group spatial keyword query*.
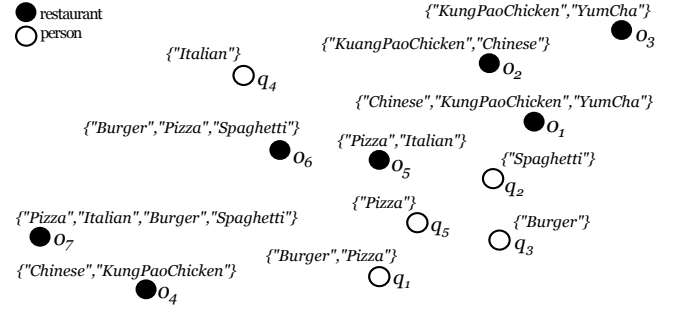

Figure 1: A set of user locations $\{q_1, q_2, q_3, q_4, q_5\}$ **and a set of restaurants** $\{o_1, o_2, ..., o_7\}$. **Restaurant** o_7 **suits the whole group the best. If size-4 subgroups are considered, then** $\{q_1, q_2, q_3, q_5\}$ **is optimal with** o_6 **being the best restaurant.**

The flexible spatial keyword query has many applications in the spatial and multimedia database domains. For example, in a *location-based social networks* (e.g., Foursquare), a group of users residing at their homes or offices can share their locations as spatial coordinates and their preferences as sets of keywords to find a Point of Interest (POI), e.g., restaurant or function venue, that optimizes a cost function composed of aggregate spatial distances and keyword similarities for the group. Since finding a POI that suits all group members might be difficult due to the diverse nature of choices, the group might prefer a result that is not optimal for the entire group, but is optimal for a subset of it. In such cases, we need to find optimal a *subgroup* of users and a POI that minimizes the cost function for the subgroup.

Figure 1 illustrates the query, where a group of five friends $\{q_1, q_2, q_3, q_4, q_5\}$ is trying to decide on a restaurant for a Sunday brunch. Each person has a location and a preferred type of food, represented by a set of keywords such as {"Burger", "Pizza"} or {"Italian"}, etc. There is a set of restaurants $\{o_1, o_2, ..., o_7\}$ to be selected from. Each restaurant also has a location and specializes in a certain type of cuisine which is represented by a set of keywords, e.g., {"Pizza", "Italian"}. Assume that a cost function $f()$ is used, which considers distance only and aggregates the total travel distance of all the query users in the group to a selected data object. As can be seen in Figure 1, o_5 is the data object closest to the group of query users overall and should be returned by the query. On a different occasion, the group of friends would like to maximize the number of

keywords in common between the group query and the POI returned by the query. If we modify the cost function now to stand for the dissimilarity between the respective keyword sets, to be denoted as $g()$, then it turns out that o_7 is the one that minimizes this function because it fully covers the keywords of the query users. Both f and g are extreme cases. In general, it is preferred to find an answer that optimizes both spatial distance and keyword set dissimilarity at the same time, which is the problem studied in this paper. Under such case, neither o_5 nor o_7 is a good query answer, as they are either not satisfying the query keywords or too far away. However, if we allow leaving out a user, say q_4, then more answer candidates become available. In particular, o_6 will become the best choice of the subgroup $\{q_1, q_2, q_3, q_5\}$, as it covers all the keywords, and is closer to the group. In fact, leaving any other query user out (e.g., q_2) would not obtain a better cost function value. Therefore, $\{q_1, q_2, q_3, q_5\}$ is the optimal subgroup of size 4 and o_6 is the corresponding optimal data point.

We observe that in many practical applications relaxing the requirement, i.e., not including all the query objects, has potential benefits in finding good quality answer. Consider a company that wants to find a suitable hotel where to hold the annual shareholder meeting. Each shareholder is identified by his location and a set of keywords describing the type of environment he would like the hotel to be located, like "metropolitan area", "resort" , "high altitude", "low altitude", etc. If the cost function to be optimized is an aggregate of the maximum distance traveled and text similarity the hotel selected maybe too far some of the shareholders. On the other hand, by omitting some travelers, the company could accommodate the rest with a shorter travel time. Similarly, in a ride-sharing service, the scheduler may want to find a car for multiple ride-sharers with certain service constraints formulated as keywords. As a third example, in a multimedia domain, one may want to find an image that matches with a subgroup of query images, where an object or query image is represented as a point (in a high-dimensional space) and a set of tag-words. Generally, one may prefer the subgroup size to be maximized, and hence, it benefits to explore the optimal solutions for different subgroup sizes.

The key challenge in processing the flexible group spatial keyword queries is how to utilize both the spatial and keyword preferences and to efficiently prune the search space. Another major challenge is how to find the optimal subgroups of various sizes in one pass over the data set. We design pruning methods based on branch and bound algorithms to process the queries. We further optimize the algorithms with the best-first search paradigm to minimize the number of data objects visited. Our contributions are as follows:

- We propose a new class of group queries in the spatial textual domain: (i) the group nearest neighbor with keywords (GNNK) query that finds the best data with respect to our cost function for the whole group, (ii) the flexible subgroup nearest neighbor with keywords (FSNNK) that finds the optimal subgroup and the corresponding best POI for a given subgroup size of size m (with $m \leq n$, the group size)and (iii) the multiple flexible subgroup nearest neighbor with keywords (MFSNNK) that returns in one pass the optimal subgroups and corresponding POIs for

all subgroups of size m, where $n' \leq m \leq n$ and n' being the minimum subgroup size.
- We propose pruning strategies based on branch and bound as well as best-first strategies for these three queries. The resultant algorithms can process the queries in a single pass over the dataset.
- We provide theoretical bounds for our algorithms, and evaluate them through an extensive experimental evaluation on real datasets. The results demonstrate the effectiveness and efficiency of the proposed algorithms.

The rest of the paper is organized as follows. Section 2 formulates the queries studied. Section 3 reviews related work. Section 4 describes the proposed algorithms. Sections 5 gives the cost analysis of algorithms. Section 6 reports the experimental results. Section 7 concludes the paper with a discussion of future work.

2 PROBLEM STATEMENT

Let D be a geo-textual dataset. Each object $o \in D$ is defined as a pair $(o.\lambda, o.\psi)$, where $o.\lambda$ is a location point and $o.\psi$ is a set of keywords. A query object q is similarly defined as a pair $(q.\lambda, q.\psi)$. Let $dist(q.\lambda, o.\lambda)$ be the spatial distance between q and o, and $similarity\_key(q.\psi, o.\psi)$ be the similarity between their keyword sets. We normalize both $dist(q.\lambda, o.\lambda)$ and $similarity\_key(q.\psi, o.\psi)$ so that their value lie between 0 and 1 (inclusive). The cost of o with respect to q is expressed in terms of their spatial distance and keyword set distance:

$$cost(q, o) = \alpha \cdot dist(q.\lambda, o.\lambda)$$
$$+ (1 - \alpha) \cdot (1 - similarity\_key(q.\psi, o.\psi))$$

Here, α is a user-defined parameter to control the preference of spatial proximity over keyword set similarity. Using $dist\_key(q.\psi, o.\psi) = 1 - similarity\_key(q.\psi, o.\psi)$, the cost function can be rewritten as:

$$cost(q, o) = \alpha \cdot dist(q.\lambda, o.\lambda) + (1 - \alpha) \cdot dist\_key(q.\psi, o.\psi)$$

We formulate the *GNNK*, *FSNNK* and *MFSNNK* queries based on $cost(q, o)$ as follows.

Definition 2.1. (*GNNK*). Given a set D of spatio-textual objects, a set Q of query objects $\{q_1, q_2, ..., q_n\}$, and an aggregate function f, the GNNK query finds an object $o_i \in D$ such that for any $o' \in D \setminus \{o_i\}$,

$$f(cost(q_j, o_i) : q_j \in Q) \leq f(cost(q_j, o') : q_j \in Q)$$

Definition 2.2. (*FSNNK*). Given a set D of spatio-textual objects, a set Q of query objects $\{q_1, q_2, ..., q_n\}$, an aggregate function f, a subgroup size m ($m \leq n$), and the set SG_m of all possible subgroups of size m, the FSNNK query finds a subgroup $sg_m \in SG_m$ and an object $o_i \in D$ such that for any $o' \in D \setminus \{o_i\}$,

$$f(cost(q_j, o_i) : q_j \in sg_m) \leq f(cost(q_j, o') : q_j \in sg_m)$$

and for any subgroup $sg'_m \in SG_m \setminus \{sg_m\}$,

$$f(cost(q_j, o_i) : q_j \in sg_m) \leq f(cost(q', o') : q' \in sg'_m)$$

Definition 2.3. (*MFSNNK*). Given a set D of spatio-textual objects, a set Q of query objects $\{q_1, q_2, ..., q_n\}$, an aggregate function f, and minimum subgroup size n' ($n' \leq n$), the MFSNNK query returns a set S of $(n - n' + 1)$ $\langle subgroup, data\ object \rangle$ pairs such that, each pair $\langle sg_m, o_m \rangle$ is the result of the *FSNNK* query with subgroup size m ($n' \leq m \leq n$).

If the users are interested in the k-best POIs then the queries can be generalized as k-GNNK, k-FSNNK and k-MFSNNK queries. These queries are straightforward extensions and the definitions are omitted. In this paper, we focus providing efficient solutions for the above queries for aggregate functions SUM ($\sum_{q_j \in Q} cost(q_j, o)$) and MAX ($\max_{q_j \in Q} cost(q_j, o)$). Without loss of generality our solutions work for any aggregate function that is monotonic (e.g., MIN). In our context, a monotonic function means, if we add more elements to the query set Q, the aggregate cost will either increase or remain the same.

3 RELATED WORK

Nearest Neighbor Queries. Nearest neighbor (NN) queries have been well studied in the spatial database community [2, 13]. The generalization of the nearest neighbor query is known as the kNN query. The depth-first (DF) [22] and the best-first (BF) [14] algorithms are commonly used to process the kNN queries. They assume the data objects to be indexed in a tree structure, e.g., the R-tree [12]. In the DF algorithm, child nodes are recursively visited according to their min_dist from the query point. Here the min_dist of a node is defined as the minimum Euclidean distance between its minimum bounding rectangle (MBR) and the query point. It gives a lower bound over the distances of the child nodes, and hence the algorithm can safely prune the nodes with min_dist greater than the distance of the nearest neighbor already retrieved. The BF algorithm maintains a priority queue of nodes to be visited. The nodes in the queue are ordered based on the min_dist. Initially the children of the root node are inserted into the priority queue. At each step, the node with the lowest min_dist is popped from the queue and its children are inserted. The algorithm returns the first k data objects popped from the queue as the kNN query answer.

Group Nearest Neighbor Queries. The group nearest neighbor (GNN) query [18] finds a data point that minimizes the aggregate distance for a group of query locations. SUM, MAX and MIN are commonly used aggregate functions. The generalization of the GNN query is the kGNN query, where k best group nearest neighbors are to be found. Several methods for processing GNN queries have been presented in [19]. Among those, the MBM algorithm is the state of the art. It visits the R-tree nodes in the order of their aggregate distance from the set of query points. The distance of the best data object retrieved so far is used as the pruning bound while visiting the nodes.

The flexible aggregate nearest neighbor (FANN) query [15] is a generalization of the GNN query. It returns the data object that minimizes the aggregate distance to any subset of ϕn query points, where n is the size of the query group and $0 < \phi \leq 1$. The query also returns the corresponding subset of query points. Two exact algorithms to process the FANN query have been proposed in [15]. The first uses a branch and bound method to restrict the search space, assuming that the data objects are indexed in an R-tree. The second uses the *threshold algorithm* [11] to find the answer.

A query similar to the FANN query called the *consensus query* [1] is the main motivation of our paper. Given a minimum subgroup size m and a set of n query points, the consensus query finds objects that minimize the aggregate distance for all subgroups with

sizes in the range $[m, n]$. A BF algorithm was proposed to process the consensus query.

The above group queries [1, 15, 19] only consider spatial proximity while a selecting data object, whereas, we consider both spatial proximity and textual similarity.

Spatial Keyword Queries. The spatial keyword query consists of a query location and a set of query keywords. A spatio-textual data object is returned based on its spatial proximity to the query location and textual similarity with the query keywords. A number of indexing structures for processing the spatial keyword query have been proposed [5, 9, 16, 21, 24, 25]. Among them, the IR-tree [9, 16] has been shown to be a highly efficient one. The IR-tree augments each node of the R-tree with an inverted file corresponding to the keyword sets of the child nodes. The WAND method [3, 7, 10] is proposed for document queries. This method is mainly designed for document retrieval and uses TF-IDF measures for document ranking. In our study, we consider both spatial and textual similarity, and use the IR-tree to index the data objects, although other spatial keyword indexes may be used as well. The WAND method in particular can be applied in the leaf level of the IR-tree to help compute the textual similarity.

A variant of the spatial keyword query, called *spatial group keyword query* has been introduced [4, 6]. It finds a group of objects that cover the keywords of a *single query* such that both the aggregate distance of the objects from the query location and the inter-object distances within the group are also minimized. Exact and approximate algorithms for three types of aggregate functions (SUM, MAX and MIN) have been presented in [4]. [8] studies the aggregate keyword routing (AKR) query (AKR). For a given set of users, an AKR query finds a route through a set of objects K that covers all users' keywords and minimizes the maximum distance travelled by any user to a meeting point p through K.

In a study parallel to ours, the group top-k spatial keyword query has been proposed recently [23]. This paper presents a branch-and-bound technique to retrieve the top-k spatial keyword objects for only one group of queries. This technique is essentially our branch-and-bound method described in Section 4.3 for the GNNK queries. As we show in our experimental evaluation (Section 6), our best-first technique always outperforms the branch-and-bound method substantially even for a single group query.

None of the existing work in the geo-textual domain addresses the problem of finding optimal subgroups and data objects in terms of spatial proximity and textual similarity, which is our main focus in this paper.

4 OUR APPROACH

This section presents our algorithms to process the GNNK, FSNNK and MFSNNK queries. The key challenge is to utilize the spatial distance and keyword preference together to constrain the search space as much as possible, since the performance of the algorithms is directly proportional to the search space (in both running time and I/O). Another challenge in the FSNNK and MFSNNK queries is to find the optimal subgroup from all possible subgroups.

4.1 Preliminaries

We use the IR-tree [9] to index our geo-textual dataset D. Other extensions of the IR-tree, such as the CIR-tree, the DIR-tree or the CDIR-tree [9] can be used as well.

The IR-tree is essentially an inverted file augmented R-tree [12]. The leaf nodes of the IR-tree contain references to the objects from dataset D. Each leaf node has also a pointer to an inverted file index corresponding to the keyword sets of the objects stored in that node. The inverted file index stores a mapping from the keywords to the objects where the keywords appear. Each node N of the IR-tree has the form ($N.\Lambda$, $N.\Psi$), where $N.\Lambda$ is the minimum bounding rectangle (MBR) that bounds the child node entries, and $N.\Psi$ is the union of the keyword sets in the child node entries.

Example 4.1. Figure 2a shows the locations of seven spatial objects $o_1, o_2, ..., o_7$. Figure 2b shows their keyword sets. The corresponding IR-tree and inverted files are not shown for space limitation. □

Objects	Keywords
$o_1.\psi$	t_1, t_2, t_7
$o_2.\psi$	t_2, t_5
$o_3.\psi$	t_2, t_7
$o_4.\psi$	t_1, t_2, t_3
$o_5.\psi$	t_5, t_6
$o_6.\psi$	t_1, t_3, t_4
$o_7.\psi$	t_1, t_3, t_4, t_6

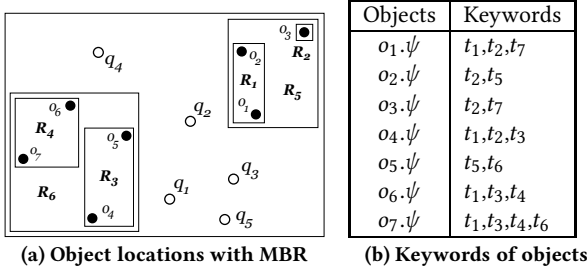**(a) Object locations with MBR**   **(b) Keywords of objects**

Figure 2: Locations and keywords of objects and queries

4.2 Cost Function

This subsection elaborates the cost function to be optimized. As defined in Section 2, the cost of an object is a combination of spatial distance and keyword dissimilarity:

$$cost(q, o) = \alpha \cdot dist(q.\lambda, o.\lambda)$$
$$+ (1 - \alpha) \cdot (1 - similarity\_key(q.\psi, o.\psi))$$

We use the Euclidean distance as the spatial distance metric. The spatial distance is normalized by the maximum spatial distance between any pair of objects in the dataset, d_{max}. Thus,

$$dist(q.\lambda, o.\lambda) = euclidean\_distance(q.\lambda, o.\lambda)/d_{max}$$

Each keyword in the dataset is associated with a weight. Following a previous study on spatial keyword search [9], we use the Language Model [20] to generate the keyword weights. The weight of each keyword is normalized by the maximum keyword weight w_{max} present in the dataset. Let $y.w$ be the weight of keyword y. Then the text relevance between q and o is the normalized sum of the weights of the keywords shared by q and o:

$$similarity\_key(q.\psi, o.\psi) = \frac{1}{|q.\psi|} \sum_{y \in q.\psi \cap o.\psi} \frac{y.w}{w_{max}}$$

Various alternative measures for textual data have been proposed, such as cosine similarity [21], the Extended Jaccard [17], etc, but extensive experiments [17] have shown that not one similarity measure outperforms the others in all cases.

Example 4.2. We continue with the example shown in Figure 2. Let the keywords of the query points be: $q_1.\psi = \{t_1, t_2\}$, $q_2.\psi = \{t_4\}$, $q_3.\psi = \{t_3, t_6\}$, $q_4.\psi = \{t_1\}$, and $q_5.\psi = \{t_4, t_6\}$. Let us assume $\alpha = 0.5$, the weight of any keyword $w = 1$ ($w_{max} = 1$), and f =SUM.

We show the aggregate cost computation for o_6. Let the distances from q_1, q_2, q_3, q_4, and q_5 to o_6 be 3.5, 5.5, 6.5, 1, and 9.5 units, and d_{max} be 10 units. Then $dist(q_3.\lambda, o_6.\lambda) = \frac{6.5}{10} = 0.65$. Meanwhile, $q_3.\psi \cap o_6.\psi = \{t_3\}$. Thus, $similarity\_key(q_3.\psi, o_6.\psi) = \frac{t3.w}{|q_3.\psi|} = 0.5$, and overall,

$$cost(q_3, o_6) = \alpha \cdot dist(q_3.\lambda, o_6.\lambda)$$
$$+ (1 - \alpha) \cdot (1 - similarity\_key(q_3.\psi, o_6.\psi))$$
$$= 0.5 * 0.65 + (1 - 0.5) * (1 - 0.5) = 0.575$$

Similarly, we compute the costs for q_1, q_2, q_4, and q_5, which are 0.175, 0.535, 0.05, and 0.725, respectively. Thus, the aggregate cost is $f(cost(Q, o_6)) = \sum_{q_j \in Q} cost(q_j, o_6) = 2.05$. □

The cost of an IR-tree node is defined similarly to the cost of a data object:

$$cost(q, N) = \alpha \, min\_dist(q.\lambda, N.\Lambda)$$
$$+ (1 - \alpha) \, (1 - similarity\_key(q.\psi, N.\Psi))$$

Here, $min\_dist(q.\lambda, N.\Lambda)$ is the minimum spatial distance between the query location and the MBR of N; $similarity\_key(q.\psi, N.\Psi)$ is the textual similarity between the query keywords and the keywords of the node. The cost of an IR-tree node gives a lower bound over the cost of its children, as formalized by the following lemma:

LEMMA 4.3. *Let N be an IR-tree node and q be a query object. If N_c is a child of N, then $cost(q, N) \leq cost(q, N_c)$.*

Proof 1. The child N_c can either be a data object or an IR-tree node. In either case $min\_dist(q.\lambda, N.\Lambda)$ is smaller than or equal to that of N_c according to the R-tree structure. Meanwhile, the keyword set of N_c is a subset of the keyword set of N. Thus, N will have a higher (or equal) textual similarity value (and hence lower keyword set distance) with the query keywords. Overall, we have $cost(q, N) \leq cost(q, N_c)$.

4.3 Branch and Bound Algorithms for GNNK and FSNNK

Traditional nearest neighbor algorithms access the data indexed in a spatial index (e.g., R-tree) and restricts its search space by pruning bounds [22]. We extend this idea to design two branch and bound algorithms for the GNNK and FSNNK queries. These two algorithms will work as the baseline algorithms in the experiments.

Branch and Bound Algorithm for *GNNK*. We use the following heuristic to prune the unnecessary nodes while searching the IR-tree for the best object with the minimum aggregate cost.

Heuristic 1. *A node N can be safely pruned if its aggregate cost with respect to the query set Q is greater than or equal to the smallest cost of any object retrieved so far.*

This heuristic is derived from Lemma 4.3. As f is a monotonic function and $cost(q, N) \leq cost(q, N_c)$ for any child N_c of N, $f(cost(Q, N))$ will be less than or equal to $f(cost(Q, N_c))$. Let min_cost be the

Algorithm 1 GNNK-BB (R, Q, f)

INPUT: IR-tree index R of all data objects, n query points $Q = \{q_1, q_2, ..., q_n\}$, monotonic cost function f.

OUTPUT: A data object o that minimizes the aggregate cost with respect to the query set Q

1: $min\_cost \leftarrow \infty$
2: $stack \leftarrow \emptyset$
3: $stack.push(root)$
4: **repeat**
5:     $N \leftarrow stack.pop()$
6:     **if** N is an intermediate node **then**
7:         **for all** N_c in $N.children$ **do**
8:             **if** $f(cost(Q, N_c)) < min\_cost$ **then**
9:                 $stack.push(N_c)$
10:     **else if** N is a leaf node **then**
11:         **for all** o in $N.children$ **do**
12:             **if** $f(cost(Q, o)) < min\_cost$ **then**
13:                 $min\_cost \leftarrow f(cost(Q, o))$
14:                 $best\_object \leftarrow o$
15: **until** $stack$ is empty
16: **return** $best\_object$

---

smallest cost of any data object retrieved so far. Then $f(cost(Q, N)) \geq min\_cost$ implies that the cost of any descendant of N is greater than or equal to min_cost, and we can safely prune N.

Algorithm 1 shows the pseudo-code of the branch and bound algorithm based on the heuristic, denoted by GNNK-BB. The algorithm maintains a stack of nodes/objects to be visited. The lowest cost object visited so far as well as the lowest cost are maintained in the variables $best\_object$ and min_cost, respectively. The algorithm starts with inserting the root node of the IR-tree into the stack (Line 3). At each step, it gets the next node/object from the stack (Line 5) and computes the aggregate query cost for each of the child nodes (if any) (Lines 7-8 and 11-12). If the child is a data object and its cost is lower than min_cost, then we update min_cost with the aggregate cost of that child (Lines 12-14). Otherwise the child is an IR-tree node and if its cost is lower than min_cost, we insert it into the stack so that we can visit its children later (Lines 8-9). At the end when the stack becomes empty, the algorithm returns the object corresponding to min_cost as the result (Line 16).

Table 1: Example of the GNNK-BB algorithm

Step	S	Elm	$f(cost)$	$best\_obj$	min_cost	S (updated)
1	$root$	$root$	$R_5 : 2.475$ $R_6 : 0.725$	$\emptyset$ $\emptyset$	∞ ∞	R_5, R_6
2	R_6, R_5	R_6	$R_3 : 1.75$ $R_4 : 1.1$	$\emptyset$ $\emptyset$	∞ ∞	R_5, R_3, R_4
3	R_5, R_3, R_4	R_4	$o_6 : 2.05$ $o_7 : 1.625$	o_6 o_7	2.05 1.625	R_5, R_3
4	R_5, R_3	R_3	$o_4 : 2.75$ $o_5 : 3.0$	o_7 o_7	1.625 1.625	R_5
5	R_5	R_5	$f(cost(Q, R_5)) > min\_cost \Rightarrow$ prune R_5; $S = \emptyset$, return o_7

Example 4.4. (**GNNK-BB**). We continue with Example 4.2. Table 1 summarizes the MFSNNK-BF steps using aggregate function SUM. Column S shows the stack; column Elm shows the element popped out; $f(cost)$ shows the aggregate costs of the child nodes;

column $best\_obj$ and min_cost shows current best object and minimum cost, respectively; column S (*updated*) shows the updated stack after processing the popped element.

At start, the tree root is popped out. The aggregate cost for each children R_5 and R_6 is less than the initialized cost ∞ and so, they are pushed into the stack. In step 3, leaf node R_4 is popped. So $best\_obj$ and min_cost are updated. In Step 4, the cost of each object o_4 and o_5 is greater than min_cost, so no update occurs. In step 5 cost of R_5 is greater than min_cost. So, R_5 is pruned, and stack S becomes empty. Then algorithm terminates, and o_7 is returned, which is the current best object. □

Branch and Bound Algorithm for *FSNNK*. We design a similar branch and bound algorithm named FSNNK-BB for the FSNNK query. The following heuristic is used for pruning.

Heuristic 2. *Let N be an IR-tree node and m be the required subgroup size. If sg_m is the best subgroup of size m, and min_cost is the smallest cost of any size-m subgroup retrieved so far, we can safely prune N if $f(cost(sg_m, N)) \geq min\_cost$.*

This heuristic is derived from Lemma 4.3. Let N_c be a child of N and sg'_m be the best subgroup corresponding to N_c. Then we have

$$f(cost(sg'_m, N)) \leq f(cost(sg'_m, N_c))$$

Meanwhile sg_m is the best subgroup for N among all possible subgroups of size m. Thus,

$$f(cost(sg_m, N)) \leq f(cost(sg'_m, N))$$

The two inequalities imply that $f(cost(sg_m, N)) \leq f(cost(sg'_m, N_c))$, i.e., the aggregate cost for the best size-m subgroup of N is lower than or equal to that of the best size-m subgroup of any of its children. Therefore, if $f(cost(sg_m, N)) \geq min\_cost$, $f(cost(sg_m, N_c))$ will also be greater than or equal to min_cost, and we should prune N.

The overall tree traversal procedure is similar to that of the GNNK-BB algorithm. The difference is in the calculation of the optimization function, where the optimization function value is computed based on the the top-m queries with the lowest costs. For an intermediate node N, we compute the best subgroup and the aggregate cost (bound) in a similar way for all of its child nodes. First, the costs from all the query points to a node are calculated. Then m query points with lowest costs are taken to get the best subgroup sg_m. If the aggregate cost for sg_m is lower than min_cost, then we insert the child node into the stack. Otherwise, it is pruned. We omit the details due to space constraints.

4.4 Best-first Algorithms for GNNK and FSNNK

Branch and bound techniques may access unnecessary nodes during query processing. To improve the query efficiency by reducing disk accesses, we propose in this section best-first search techniques that only access the necessary nodes.

Best-first algorithm for GNNK. The best-first procedure for the GNNK query, denoted by GNNK-BF, is shown in Algorithm 2. This algorithm uses a minimum priority queue P to maintain the nodes/objects to be visited according to their aggregate costs. At start, the queue P is initialized with the root of the IR-tree (Lines 1-2). At each iteration of the main loop (Lines 3-13), the element with the minimum aggregate cost is popped out from P. There

Algorithm 2 GNNK-BF (R, Q, f)

INPUT: IR-tree index R of all data objects, n query points $Q = \{q_1, q_2, ..., q_n\}$, monotonic cost function f.

OUTPUT: A data object o that minimizes the aggregate cost with respect to the query set Q

1: Initialize a new min priority queue P
2: $P.push(root, 0)$
3: **repeat**
4:     $E \leftarrow P.pop()$
5:     **if** E is an intermediate node N **then**
6:         **for all** N_c in $N.children$ **do**
7:             $P.push(N_c, f(cost(Q, N_c)))$
8:     **else if** E is a leaf node N **then**
9:         **for all** o in $N.children$ **do**
10:            $P.push(o, f(cost(Q, o)))$
11:     **else if** E is a data object o **then**
12:         return o
13: **until** P is empty
14: return $null$

are three cases to be considered for a popped element: *(i)* If it is an intermediate node, then all child nodes are pushed into P according to their aggregate costs (Lines 5-7). *(ii)* If it is a leaf node, then all child objects are pushed into P according to their aggregate costs (Lines 8-10). *(iii)* If it is an object, then it is returned as the query result (Lines 11-12), and the algorithm terminates (Line 14).

Example 4.5. (**GNNK-BF**). We continue with Example 4.2. The algorithm steps are summarized in Table 2, where SUM is used as the aggregate function. Column P shows the current elements in the queue; column *Element* shows the element popped out in the current step; column $f(cost)$ shows the aggregate costs of the child nodes of the popped element; column P (*updated*) shows the updated queue after processing the popped element.

Table 2: Example of the GNNK-BF algorithm

Step	P	Element	$f(cost)$	P (updated)
1	$root$	$root$	$R_5 : 2.475$ $R_6 : 0.725$	R_6, R_5
2	R_6, R_5	R_6	$R_3 : 1.75$ $R_4 : 1.1$	R_4, R_3, R_5
3	R_4, R_3, R_5	R_4	$o_6 : 2.05$ $o_7 : 1.625$	o_7, R_3, o_6, R_5
4	o_7, R_3, o_6, R_5	o_7	$return \quad o_7$	

At start, the tree root is popped out. The aggregate costs for the children R_5 and R_6 are computed and they are pushed into the queue. The node R_6 has the lowest aggregate cost, and hence it is at the front of the queue. In the next step, R_6 is popped out and the aggregate costs for its children R_3 and R_4 are computed. This procedure repeats until Step 4 where o_7 is popped out. This is the first data object popped out. According to the algorithm, this object is the best object for the query, and hence it is returned as the query answer. □

LEMMA 4.6. *(**Proof of Correctness**) GNNK-BF returns the object with the minimum aggregate cost w.r.t. the query set Q.*

Proof 2. Let o be the data object returned by GNNK-BF, i.e, o is the first data object visited by the algorithm. Assume that a different object $o′$ is the data object with the minimum aggregate cost. Then $f(cost(Q, o′)) \leq f(cost(Q, o))$. Let N be the first common ancestor of o and $o′$ in the IR-tree. We know from Lemma 4.3 that the cost of an IR-tree node gives a lower bound over the costs of its children. Thus, any node in the path from N to the parent of $o′$ will have a lower aggregate cost than that of $o′$. This implies that these nodes have lower aggregate costs than that of o, and should be visited before o. Therefore, when o is visited, $o′$ must be in the priority queue as its parent has already been visited. Because $o′$ has a lower cost than o has, it should be visited first, which means that $o′$ must be the data object returned by GNNK-BF rather than o. This is conflict to our assumption, and hence $o′$ should not have been existed. Therefore, o must be the data object with the minimum aggregate cost.

Algorithm 3 FSNNK-BF (R, Q, m, f) [partial]

1: ...
2: **if** E is an intermediate node N **then**
3:     **for all** N_c in $N.children$ **do**
4:         Compute $cost(q_1, N_c), ..., cost(q_n, N_c)$
5:         $sg_m \leftarrow$ first m query points with the lowest costs
6:         $P.push(N_c, f(cost(sg_m, N_c)))$
7: **else if** E is a leaf node N **then**
8:     ...
9: **else if** E is a data object o **then**
10:     return $(o, o.best\_subgroup)$
11: ...

Best-first Algorithm for FSNNK. The best-first algorithm for the FSNNK query, denoted by FSNNK-BF, is similar to GNNK-BF algorithm. This algorithm also maintains a minimum priority queue to manage the nodes/objects to be visited from the IR-tree, and traverses the tree from the root. Here, optimization function is computed for top-m queries. Best subgroup is chosen from the lowest m query points, and pushed into the priority queue. For an intermediate node, aggregate costs and best subgroup are calculated for all the child nodes of the node. For a leaf node, it is done for all the children objects, and then pushed into the priority queue. When an object is first popped, it is returned as the result. The partial pseudo-code is shown in Algorithm 3.

Example 4.7. (**FSNNK-BF**). We continue with Example 4.2 for the FSNNK query. Let the subgroup size $m = 3$. The algorithm steps for FSNNK-BF are summarized in Table 3. Column *Element* shows the elements popped out from P at every step; column sg_m shows the best subgroup of size m corresponding to the current node or data object, which is also the set of m lowest cost query points corresponding to the current node or data object; column $f_m(cost)$ is the aggregate cost over the query points in sg_m; column P (*updated*) shows the updated queue after processing the popped element.

At start, the tree root is popped out. The individual costs for children R_5 and R_6 are computed. The best subgroups for R_5 and R_6 are shown in the sg_m column. The aggregate costs for R_5 and R_6 are also computed. Both nodes are then pushed into P. In the

Table 3: Example of the FSNNK-BF algorithm

Step	Element	$f_m(cost)$	sg_m	P (updated)
1	$root$	$R_5 : 1.2$ $R_6 : 0.225$	q_4, q_1, q_2 q_4, q_1, q_2	R_6, R_5
2	R_6	$R_3 : 0.45$ $R_4 : 0.475$	q_1, q_4, q_3 q_4, q_1, q_2	R_3, R_4, R_5
3	R_3	$o_4 : 1.15$ $o_5 : 1.7$	q_1, q_4, q_3 q_3, q_1, q_5	R_4, o_4, R_5, o_5
4	R_4	$o_6 : 0.75$ $o_7 : 0.8$	q_4, q_1, q_2 q_1, q_4, q_3	o_6, o_7, o_4, R_5, o_5
5	o_6	$return$	$(o_6, \{q_1, q_4, q_3\})$	

next step, R_6 is popped out, as it has the minimum cost. The computation for the children of R_6 is carried out in the same way. This procedure repeats, and at Step 5, o_6 is popped out. It is the first object popped out, which gives the minimum aggregate cost among all data objects. FSNNK-BF returns o_6 and the corresponding best subgroup $\{q_1, q_4, q_3\}$ as the query answer. □

4.5 Algorithms for MFSNNK

Algorithm 4 MFSNNK-BF (R, Q, m, f)

INPUT: IR-tree index R of all data objects, n query points $Q = \{q_1, q_2, ..., q_n\}$, minimum subgroup size $m(m \leq n)$, monotonic cost function f.

OUTPUT: A set of $\langle data\_object, subgroup \rangle$ pairs $\langle o_k^*, sg_k^* \rangle$ for all subgroup sizes between m and n (inclusive), where $\langle o_k^*, sg_k^* \rangle$ minimizes $f(cost(sg_k, o))$.

1: Initialize a new min priority queue P
2: $min\_costs[i] \leftarrow \infty$ for $m \leq i \leq n$
3: $root.query\_costs[i] \leftarrow 0$ for $m \leq i \leq n$
4: $P.push(root, 0)$
5: **repeat**
6:    $E \leftarrow P.pop()$
7:    **if** $\exists i \in [m, n]: E.query\_costs[i] < min\_costs[i]$ **then**
8:       **if** E is an intermediate node **then**
9:          **for all** N_c in $E.children$ **do**
10:             Compute $cost(q_1, N_c), ..., cost(q_n, N_c)$
11:             $total\_cost \leftarrow 0$
12:             **for** $i = m \rightarrow n$ **do**
13:                $sg_i \leftarrow$ top i lowest cost query points
14:                $total\_cost += f(cost(sg_i, N_c))$
15:                $N_c.query\_costs[i] = f(cost(sg_i, N_c))$
16:             **if** $f(cost(sg_i, N_c)) < min\_costs[i]$ for any subgroup size $i \in [m, n]$ **then**
17:                $P.push(N_c, total\_cost)$
18:       **else if** E is a leaf node **then**
19:          **for all** o in $N.children$ **do**
20:             Compute $cost(q_1, o), ..., cost(q_n, o)$
21:             **for** $i = m \rightarrow n$ **do**
22:                $sg_i \leftarrow$ top i lowest cost query points
23:                **if** $f(cost(sg_i, o)) < min\_costs[i]$ **then**
24:                   $min\_costs[i] \leftarrow f(cost(sg_i, o))$
25:                   $best\_objects[i] \leftarrow o$
26:                   $best\_subgroups[i] \leftarrow sg_i$
27: **until** P is empty
28: **return** $best\_objects, best\_subgroups$

To process the MFSNNK query with a minimum subgroup size m, we can run FSNNK-BF $n - m + 1$ times (for subgroup sizes $m, m + 1, ..., n$) and return the combined results. We call this the MFSNNK-N algorithm. However, MFSNNK-N requires accessing the dataset $n - m + 1$ times, which is too expensive. To avoid this repeated data access, we design an algorithm based on best-first method that can find the best data objects for all subgroup sizes between m and n in a single pass over the dataset. The algorithm is based on the following heuristic.

Heuristic 3. *Let N be an IR-tree node and m be the minimum subgroup size. Let sg_i be the best subgroup of size i ($m \leq i \leq n$), and min_cost_i be the smallest cost for subgroup size i from any object retrieved so far. We can safely prune N if $f(cost(sg_i, N)) \geq min\_cost_i$ for any i.*

The proof of correctness is straightforward based on Heuristic 1 and Heuristic 2, and is omitted due to space limit.

Algorithm 4 summarizes the proposed procedure, denoted as MFSNNK-BF. The algorithm maintains a minimum priority queue P to manage the nodes/objects to be visited from the IR-tree (Line 1). The minimum costs for all subgroup sizes in the range $[m, n]$ are set to ∞ at the beginning (Line 2). Each tree node to be visited is associated with an array $query\_costs$ that keeps track of the aggregate costs for all subgroup sizes in the range $[m, n]$ (Line 3). The algorithm pushes the tree root into the queue P and then the main loop begins (Lines 6-36). At each iteration, an element is popped out from P. The associated $query\_costs$ (already computed at a previous iteration) is compared with min_costs. If $query\_cost$ for any subgroup size is lower than the min_cost of that subgroup size, the element needs to be considered further. Otherwise the element is pruned according to Heuristic 3 (Lines 7-8). There are two cases to be further considered: *(i)* If the element is an intermediate node, then we compute the costs for each child node (Lines 10-11). We compute the aggregate cost for each subgroup size in the range $[m, n]$ (Lines 13-17), and store the corresponding best query subgroup in sg_i. If the aggregate cost is larger than min_cost for all subgroup sizes, the child node can be safely pruned. Otherwise we insert the child node into P according to its total cost. (Lines 18-20) *(ii)* If the element is a leaf node, then a similar computation is performed for each child object (Lines 23-26). If the aggregate cost is less than min_cost for a subgroup size i, then $min\_costs[i]$, $best\_objects[i]$, and $best\_subgroups[i]$ are updated (Lines 27-31).

Table 4: Example of the MFSNNK-BF algorithm

Step	Elm	$m = 3, m = 4, m = 5$	$best\_obj$	min_costs	$total\_cost$	P (updated)
1	$root$	$R_5 : 1.2, 1.775, 2.475$ $R_6 : 0.225, 0.45, 0.725$	$\{\emptyset, \emptyset, \emptyset\}$ $\{\emptyset, \emptyset, \emptyset\}$	$\{\infty, \infty, \infty\}$ $\{\infty, \infty, \infty\}$	5.45 1.4	R_6, R_5
2	R_6	$R_3 : 1.2, 1.075, 1.75$ $R_4 : 0.225, 0.775, 1.1$	$\{\emptyset, \emptyset, \emptyset\}$ $\{\emptyset, \emptyset, \emptyset\}$	$\{\infty, \infty, \infty\}$ $\{\infty, \infty, \infty\}$	4.025 2.13	R_4, R_3, R_5
3	R_4	$o_6 : 0.75, 1.325, 2.05$ $o_7 : 0.8, 1.125, 1.625$	$\{o_6, o_6, o_6\}$ $\{o_6, o_7, o_7\}$	$\{0.75, 1.325, 2.05\}$ $\{0.75, 1.125, 1.625\}$	$\emptyset$ $\emptyset$	R_3, R_5
4	R_3	$o_4 : 1.15, 1.9, 2.75$ $o_5 : 1.7, 2.325, 3.0$	$\{o_6, o_6, o_7\}$ $\{o_6, o_6, o_7\}$	$\{0.75, 1.125, 1.625\}$ $\{0.75, 1.125, 1.625\}$	$\emptyset$ $\emptyset$	R_5
5	R_5	$R_5.query\_costs[i] > min\_costs[i]$ for all $m = i$, and hence R_5 can be pruned				

Example 4.8. (**MFSNNK-BF**). We continue with Example 4.2 for MFSNNK-BF. Let the minimum subgroup size be 3. Then we need to find the best objects and the corresponding subgroups for $m = 3$, $m = 4$, and $m = 5$. The algorithm steps are shown in Table 4.

Column *Elm* shows the elements popped out from the queue P. The following column show the aggregate costs ($f_m(cost)$) for different subgroup sizes. $total\_cost$ is the sum of $f_m(cost)$ for all subgroup sizes.

At start, min_costs is initialized with value ∞, and *root* is pushed into P. Then nodes are popped and calculations are performed as shown in Step 1 and Step 2. At Step 3, R_4 is popped out, which is a leaf node. The algorithm updates min_costs, $best\_objects$ and the subgroup set $best\_subgroups$ as R_4 has objects as children. When a object is popped, $best\_objects$ are updated according to min_costs. When P becomes empty after Step 5, the algorithm returns $(o_6, \{q_4, q_1, q_2\})$, $(o_7, \{q_4, q_1, q_3, q_2\})$, $(o_7, \{q_4, q_1, q_3, q_2, q_5\})$. □

A relaxed pruning bound. Heuristic 3 states that, for an IR-tree node N, if min_cost_i is the smallest cost for subgroup size i found so far, then we can prune N if $f(cost(sg_i, N)) \geq min\_cost_i$ for any $i \in [m..n]$. Here, sg_i denotes the best subgroup of size i corresponding to N. The MFSNNK-BF algorithm based on this heuristic has a for-loop to compute $f(cost(sg_i, N))$ and test if $f(cost(sg_i, N)) \geq min\_cost_i$ holds for any i (Lines 12 to 17 in Algorithm 4).

A possible simplification is to only test whether $f(cost(sg_m, N)) \geq min\_cost_n$, i.e., whether the best subgroup of size m corresponding to N has a cost lower than the min_cost for the whole group of size n found so far. If this holds, then N can be safely pruned, as formalized by the following heuristic.

Heuristic 4. *Let N be an IR-tree node and m be the minimum subgroup size. Let sg_m be the best subgroup of size m corresponding to N, and min_cost_n be the smallest cost for the whole group of size n from any object retrieved so far. We can safely prune N if $f(cost(sg_m, N)) \geq min\_cost_n$.*

The proof is straightforward. Since we consider a monotonic aggregate cost function, we have:

$$f(cost(sg_m, N)) \leq f(cost(sg_{m+1}, N)) \leq ... \leq f(cost(sg_n, N)).$$

If

$$f(cost(sg_m, N)) \geq min\_cost_n,$$

then

$$min\_cost_n \leq f(cost(sg_m, N)) \leq ... \leq f(cost(sg_n, N)).$$

Thus, we can safely prune N. Applying this heuristic, Lines 12 to 17 of Algorithm 4 can be replaced by:

$$\text{If } f(cost(sg_m, N_c)) < min\_costs[n] \text{ then}$$
$$P.push(N_c, f(cost(sg_m, N_c)))$$

Note that, while this heuristic simplifies the node pruning computation, it also relaxes the pruning bound, which may cause more nodes to be processed. We will use experiments to study the effectiveness of this heuristic.

4.6 Discussion

All the algorithms presented in the previous subsections can be straightforwardly extended to find the k best objects. Both the GNNK-BF and FSNNK-BF algorithms incrementally output the best objects. The first k objects accessed by these algorithms are the k best objects. Particularly, in the case of FSNNK-BF, we can use a queue to store the k best objects and the corresponding best

subgroups. For the GNNK-BB, FSNNK-BB and MFSNNK-BF algorithms, we can use a heap of size k to hold k currently found best objects. When the algorithms terminate, the heap contains the k best objects. Same as in FSNNK-BF, for the subgroup queries we can store the best objects and the corresponding best subgroups together, so that when the algorithms terminate, we not only obtain the best objects but also the corresponding best subgroups.

Though in our problem formulation, we assume that all users in the group have equal priorities, our proposed cost function can be adapted for users with different priorities. Assume that each individual query q_i has a priority p_i associated with it, where for a group of n queries $p_1 + p_2 + ... + p_n = n$. To incorporate user priorities, we need to modify our definition of aggregate cost function as follows: $f(cost(Q, o)) = f(\frac{cost(q_j, o)}{p_j} : q_j \in Q)$. Thus, users with higher priorities (i.e., larger priority values p_i) would have lower costs, and hence the algorithms will tend to converge more to the objects that are spatially closer and textually more similar to the users with higher priority.

5 COST ANALYSIS

We analytically compare the I/O cost and CPU cost of the algorithms including GNNK-BB, GNNK-BF, FSNNK-BB, FSNNK-BF, MFSNNK-N, and MFSNNK-BF. Table 5 summarizes the analytical results. Note that MFSNNK-N calls FSNNK-BF for $n - m + 1$ times. Its costs are just a multiplication of those of FSNNK-BF. We will omit it in the discussion and simply list its costs in the table.

We use the following notation in the analysis. Let C_m be the maximum number of entries in a disk block:

$$C_m = \text{block size/size of a data entry}$$

Let C_e be the effective capacity of the IR-tree used to index the dataset D, i.e., the average number of entries in an IR-tree node. Let $|D|$ be the size of D. The average height of an IR-tree is $h = \lceil \log_{C_e} |D| \rceil$. The expected number of nodes in an IR-tree is the total number of nodes in all tree levels (leaf nodes being level 1 and the root node being level h), which is:

$$\sum_{i=1}^{h} \frac{|D|}{C_e^i} = |D| \left(\frac{1}{C_e} + \frac{1}{C_e^2} + \cdots + \frac{1}{C_e^h} \right) = \frac{|D|}{C_e - 1}(1 - \frac{1}{C_e^h}) \approx \frac{|D|}{C_e - 1}.$$

We assume that an IR-tree node size equals a disk block.

According to the structure of the IR-tree, an inverted index that maps keywords to the inner nodes of the tree is stored separately from the tree structure. When a group spatial keyword query is issued this inverted index is preloaded for all the query keywords, which will be used to guide the search to tree nodes that contain the query keywords. The cost of this preloading, which is proportional to the number of keywords in both the data points and the queries, is the same for every algorithm studied. We denote the I/O cost and CPU cost of the preloading by io_i and cpu_i, respectively.

5.1 I/O Cost

For all the algorithms studied, the I/O costs depend on the number of IR-tree nodes accessed. Further, when a leaf node is accessed, its corresponding inverted index that maps the keywords to the data points in the node is accessed as well. Analyzing the I/O cost of accessing an inverted index is beyond the scope of this paper. For

Table 5: Summary of Costs

Algorithm	I/O	CPU
GNNK-BB	$io_i + (1 - w_{gb})(\frac{|D|}{C_e - 1} + \frac{|D|}{C_e} \cdot io_l)$	$cpu_i + (1 - w_{gb})(\frac{|D|}{C_e - 1} \cdot cpu_g + \frac{|D|}{C_e} \cdot cpu_l)$
GNNK-BF	$io_i + (1 - w_{gf})(\frac{|D|}{C_e - 1} + \frac{|D|}{C_e} \cdot io_l)$	$cpu_i + (1 - w_{gf})(\frac{|D|}{C_e - 1} \cdot cpu_g + \frac{|D|}{C_e} \cdot cpu_l)$
FSNNK-BB	$io_i + (1 - w_{sb})(\frac{|D|}{C_e - 1} + \frac{|D|}{C_e} \cdot io_l)$	$cpu_i + (1 - w_{sb})(\frac{|D|}{C_e - 1} \cdot cpu_s + \frac{|D|}{C_e} \cdot cpu_l)$
FSNNK-BF	$io_i + (1 - w_{sf})(\frac{|D|}{C_e - 1} + \frac{|D|}{C_e} \cdot io_l)$	$cpu_i + (1 - w_{sf})(\frac{|D|}{C_e - 1} \cdot cpu_s + \frac{|D|}{C_e} \cdot cpu_l)$
MFSNNK-N	$io_i + (n - m + 1)(1 - w_{sf})(\frac{|D|}{C_e - 1} + \frac{|D|}{C_e} \cdot io_l)$	$cpu_i + (n - m + 1)(1 - w_{sf})(\frac{|D|}{C_e - 1} \cdot cpu_s + \frac{|D|}{C_e} \cdot cpu_l)$
MFSNNK-BF	$io_i + (1 - w_{mb})(\frac{|D|}{C_e - 1} + \frac{|D|}{C_e} \cdot io_l)$	$cpu_i + (1 - w_{mb})(\frac{|D|}{C_e - 1} \cdot cpu_m + \frac{|D|}{C_e} \cdot cpu_l)$

simplicity, we denote this I/O cost by io_l, and the associated CPU cost by cpu_l.

GNNK-BB, GNNK-BF, FSNNK-BB, FSNNK-BF, and MFSNNK-BF all traverses the IR-tree for only once. In the worst case, all the tree nodes plus the inverted index of all the leaf nodes are accessed. Thus, the worst-case I/O costs for these methods are the same: $\frac{|D|}{C_e - 1} + \frac{|D|}{C_e} \cdot io_l$.

In the average case, some of the IR-tree nodes are pruned during the traversal. We quantify the percentage of pruned nodes in the tree traversal as the pruning power, denoted by w; the number of nodes accessed is then $(1 - w)(\frac{|D|}{C_e - 1} + \frac{|D|}{C_e} \cdot io_l)$ for all the algorithms except FSNNK-BF, where w should be replaced by w_{gb}, w_{gf}, w_{sb}, w_{sf}, and w_{mb} for GNNK-BB, GNNK-BF, FSNNK-BB, FSNNK-BF, and MFSNNK-BF, respectively. Note that we use w to represent the pruning power on both inner nodes and leaf nodes, which might be different in reality. We argue that this is still a reasonable simplification since the number of leaf nodes pruned will be proportional to the number of inner nodes pruned. Also we aim to compare the costs of the different algorithms, not to compute the exact costs.

The pruning power of the different algorithms is associated with the metrics used to determine whether a tree node needs to be accessed. In the algorithms studied, the same pruning metric (e.g., min_cost) is used for different algorithms of the same query variant (GNNK-BB and GNNK-BF for the GNNK query). However, the order that the tree nodes are accessed in the different algorithms of the same query variant (e.g., GNNK-BB and GNNK-BF) are different. This leads to different shrinking rates of the value of the pruning metric. In particular, the BF algorithms and MFSNNK-BF use best-first traversals, which always access the node with the smallest (estimated) optimization function value first. In comparison, the BB algorithms simply push the tree nodes into a stack, and access the node at the top of the stack regardless of the optimization function value. Heuristically, the BF algorithms' pruning metric values should shrink faster. Additionally, the BF algorithms terminates early once a data entry is popped out from the queue, while the BB algorithms need to access every node in the stack anyway. Intuitively, the BF algorithms should have better pruning power than those of the corresponding BB algorithms, i.e, $w_{gf} > w_{gb}$ and $w_{sf} > w_{sb}$. MFSNNK-BF only traverses the tree once, and it has a similar pruning strategy to that of FSNNK-BF. Its I/O cost is smaller than that of MFSNNK-N that calls FSNNK-BF multiple times.

5.2 CPU Cost

The CPU cost can be considered as the product of the CPU cost per block (node) multiplied by the number of blocks (nodes) accessed. The I/O cost analysis provides the number of nodes accessed. The CPU cost per block, denoted by cpu, involves optimization function computation.

Both GNNK algorithms computes $f(cost(Q, N_c))$ for every child node N_C when an inner node N accessed, and $f(cost(Q, o))$ for every data point o if N is a leaf node. The CPU cost is proportional to the size of the query group (n), the size of the node N (C_e), and the size of the keywords involved. Sine this per node CPU cost of both GNNK-BB and GNNK-BF is the same, we simply denote it by cpu_g. Note that GNNK-BF still has a lower overall CPU cost as it accesses a smaller number of nodes.

Similarly, we denote the per node CPU cost of FSNNK-BB and FSNNK-BF by cpu_s. This CPU cost involves computing $cost(q_i, N_c)$ ($cost(q_i, o)$) for every query user q_i, finding to top-m users, and computing $f()$ on the cost of these m users. FSNNK-BF also has a lower overall CPU cost as it accesses a smaller number of nodes.

MFSNNK-N has the same per node CPU cost cpu_s. Let the per node CPU cost of MFSNNK-BF be cpu_m. This cost will be higher than cpu_s as MFSNNK-N only computes the optimization function value of a given subgroup size each time it access a node, while MFSNNK-BF computes for $n - m + 1$ subgroup sizes together. However, $cpu_m < (n - m + 1)cpu_s$. This is because, as shown in lines 11 to 17 of the MFSNNK-BF algorithm, the functions $cost(q_i, N_c)$ are computed for only once rather than $n - m + 1$ times, and the function $f()$ for the different sub-group size are computed progressively instead of repeatedly. As a result, the overall CPU cost of MFSNNK-BF will be lower than that of MFSNNK-N.

6 EXPERIMENTAL EVALUATION

6.1 Experimental Settings

We evaluate the performance of our algorithms for all three types of queries GNNK, FSNNK, and MFSNNK. The branch and bound algorithms presented in Section 4.3 (GNNK-BB and FSNNK-BB) are used as the baseline for the GNNK and FSNNK queries. We compare our BF algorithms, GNNK-BF and FSNNK-BF (Section 4.4) with baselines. We use the MFSNNK-N algorithm as the baseline algorithm for the MFSNNK queries, and compare it with the MFSNNK-BF algorithm proposed in Section 4.5.

Table 6: Dataset properties

Parameter	Flickr	Yelp
Dataset size	1,500,000	60,667
Number of unique keywords	566,432	783
Total number of keywords	11,579,622	176,697
Avg. number of keywords per object	7.72	2.91

Table 7: Query parameters

Parameter name	Values	Default Value
Number of queried data points (k)	1, 10, 20, 30, 40, 50	10
Query group size (n)	10, 20, 40, 60, 80	10
Subgroup size (m, %n)	40%, 50%, 60%, 70%, 80%	60%
Number of query keywords	1, 2, 4, 6, 8, 10	4
Size of the query space	.001%, .01%, .02%, .03%, .04%, .05%	0.01%
Size of the query keyword set	1%, 2%, 3%, 4%, 5%	3%
Spatial vs. textual preference (α)	0.1, 0.3, 0.5, 0.7, 1.0	0.5
Dataset Size (Flickr)	1M, 1.5M, 2M, 2.5M	1.5M

Dataset. We use two real datasets from Yahoo! Flickr[1] and Yelp[2] in our experiments. The Flickr dataset is generated from the images from Yahoo! Flickr users that are geo-tagged and contain a set of keyword tags. The Yelp dataset contains the basic information about different local businesses. Each data object contains the location of the business along with its categories as the keywords. Only the data locations within US have been used in our experiment. The properties of these two datasets are detailed in Table 6.

Query Generation. We generate 20 groups of query objects for each experiment and average the results. Each query object contains a location and a set of keywords. To generate the locations in each group of query objects, we first randomly choose a point in the data space. Then we define a square query space centered at the chosen point. All the query object locations of the group will then be uniformly generated inside this square query space. The default query space area has been selected to be 0.01% (250 sq. miles) of the total query area, which is approximately the size of a medium sized US city. Similarly, for generating the query keywords, a subset of keywords (1%-5% of the data objects' keywords) from all keywords inside the query space is first chosen, and then the required of number of keywords are selected from this subset. This ensures the overlapping of query keywords among users.

We also vary the group size (n), the minimum subgroup size (m), the number of query keywords, the number of queried data points (k), dataset size, and α. Table 7 shows ranges and default values of these parameters.

Setup. We use the IR-tree to index the datasets, which is disk resident. The fanout of the IR-tree is chosen to be 50, and the page size is 4KB. All the algorithms are implemented in Java and the experiments are conducted on a Core i7-4790 CPU @ 3.60 GHz with 4 GB of RAM. The hard drive used is Seagate ST500DM002-1BD142 with 7200 RPM. SUM and MAX are used as the aggregate functions in all the experiments.

We measure the running time and the I/O cost (number of disk page accesses) in the experiments. Note that the running time includes the computation and I/O time. We use Flickr as our default dataset, unless stated otherwise.
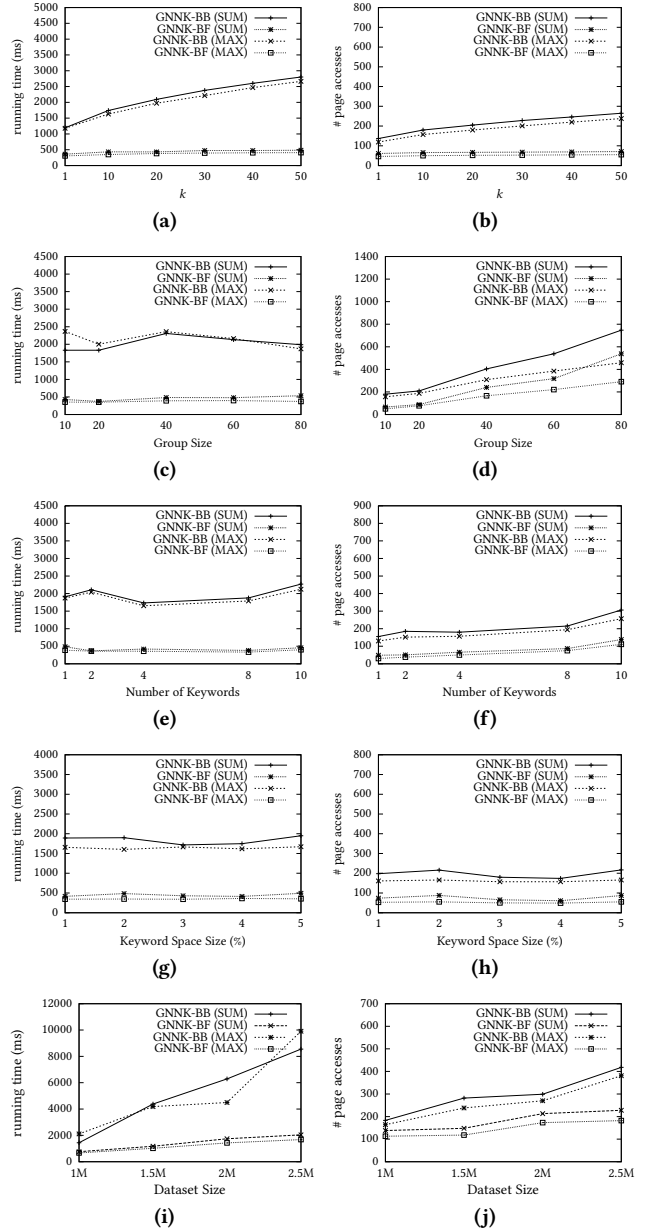
---

[1] https://webscope.sandbox.yahoo.com
[2] https://www.yelp.com/academic_dataset

Figure 3: The effect of varying k (a-b), query group size (c-d), number of query keywords (e-f), query keyword set size (g-h) and dataset size (i-j) in running time and I/O

6.2 The GNNK Query Algorithms

We conduct seven sets of experiments to evaluate the performance of GNNK-BB and GNNK-BF. In each set of experiments, one parameter (e.g., group size n or α) is varied while all other parameters are set to their default values. GNNK-BF outperforms GNNK-BB in all experiments both in terms of running time and I/O cost.

Varying k. Figure 3 (a-b) shows that for both GNNNK-BB and GNNK-BF, the processing time and the I/O cost increase with the increase of k. For both SUM and MAX, on average GNNK-BF runs 3.5 times faster than GNNK-BB. We also observe that for a larger

value of k, GNNK-BF algorithm outperforms GNN-BB in a greater margin, which shows the scalability of GNNK-BF. The I/O cost of GNNK-BF is much less than that of GNNK-BB as GNNK-BF only accesses the necessary nodes.

Varying Query Group Size. Figure 3 (c-d) shows the effect of the query group size (n). The query processing costs of both algorithms increase as the value of n increases. On average, GNNK-BF runs approximately 4 times faster than GNNK-BB.

Varying Number of Query Keywords. Figure 3 (e-f) shows the effect of the number of keywords in each query object. GNNK-BF again outruns GNNK-BB in all the experiments. Also, the query processing costs of both algorithms increase as the number of keywords in each query object increases. This can be explained by that a larger set of query keywords takes more time to compute the aggregate cost function. Meanwhile, more data objects' keyword sets would overlap with the query keywords, which would reduce the aggregate cost function values and make it more difficult to prune the data objects.

Varying Query Space Size. We observe that the running time of our algorithms remains almost constant with the change of the query space area (not shown in graphs). Since varied query space areas are insignificant in compared to the data space, we do not observe any significant change in this experiment.

Varying Query Keyword Set Size. Figure 3 (g-h) shows the effect of the query keyword set size (the subset of keywords from where the query keywords are generated). We see that the running time of our algorithms do not follow any regular pattern with the change of the query keyword set size and remains relatively stable.

Varying α. We observe that, as α increases, the query costs decrease. A larger α means that spatial proximity is deemed more important than textual similarity. When α increases, the impact of the keyword similarity becomes smaller and algorithms converge faster (not shown in graphs).

Varying Dataset Size. Figure 3 (i-j) shows the effect of varying number of objects. Both running time and I/O cost of our proposed algorithms increase at a lower rate than the baseline algorithms. When the number of data objects increases from 1M to 2.5M, the running time of GNNK-BB increases 6 times for SUM and 4.7 times for MAX. But the increase in running time of GNNK-BF is only 2.7 times for SUM and 2.5 times for MAX.

6.3 The FSNNK Query Algorithms

We performed experiments on FSNNK-BB and FSNNK-BF, by varying query group size, subgroup size, number of query keywords, query space size, query keyword set size, k, dataset size, and α. FSNNK-BF outperforms FSNNK-BB in all the experiments. For space constraints, we only show the effect of varying the subgroup size (in % n) in Figure 4 (a-b). On average, FSNNK-BF runs 3.5 times faster and takes 40% less I/O than FSNNK-BB.

6.4 The MFSNNK Query Algorithms

We performed similar experiments on MFSNNK-N and MFSNNK-BF. In all the experiments MFSNNK-BF significantly outperforms MFSNNK-N. Due to space constraints, we only show the effect of
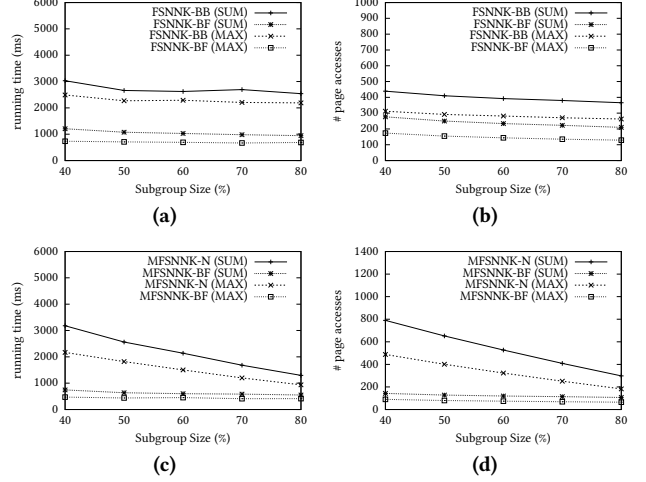

(a)   **(b)**

(c)   **(d)**

Figure 4: The effect of varying subgroup size m (a-b) and minimum subgroup size (c-d) in running time and I/O

varying the minimum subgroup size (in percentage of n) in Figure 4 (c-d). When the minimum subgroup size increases, the running time of both algorithms decrease as expected. Meanwhile, the costs of MFSNNK-BF change in a much smaller scale, which demonstrates the better scalability of MFSNNK-BF. On average, MFSNNK-BF runs about 4 times faster than MFSNNK-N.
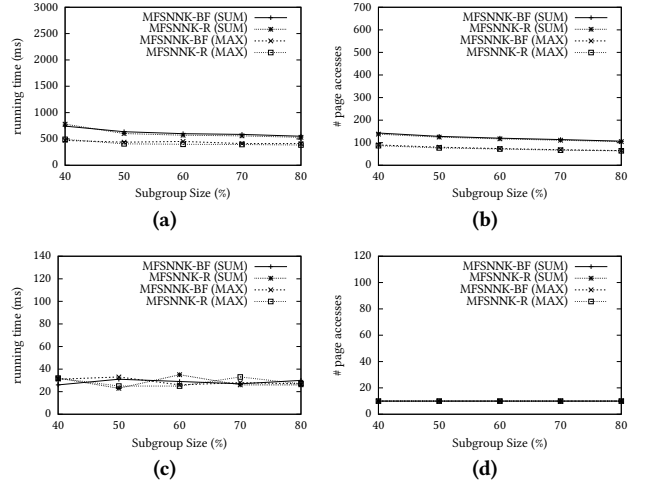

(a)   **(b)**

(c)   **(d)**

Figure 5: The effect of varying subgroup size for Flickr (a-b) and Yelp (c-d) with and without using Heuristic 4 in running time and I/O

Effect of Heuristic 4. We have also implemented Heuristic 4 for the MFSNNK query. In Figure 5, we show the performance of MFSNNK with and without using Heuristic 4, denoted by MFSNNK-R and MFSNNK-BF, respectively. We can see that, when different values of m or different data sets are used, the algorithm may perform better or worse with Heuristic 4. Particularly, on the Flikr data set, the algorithm with the pruning heuristic works better when $40\%n < m < 80\%n$, and worse when $m \leq 40\%n$ or $m \geq 80\%n$

(shown in Figure 5 (a-b)). On the Yelp data set, the algorithm performance fluctuates more but the algorithm still performs better with the pruning heuristic in about half of the cases tested (shown in Figure 5 (c-d)). This is expected since the heuristic sacrifices the tightness of the pruning bound for a more efficient computation of the pruning bound, as discussed in Section 4.5. Depending on different data sets and/or different values of m, this sacrifice may or may not be worthy. We would like to argue that, however, since the data sets are usually pre-known, we may empirically pre-test the heuristic performance under a set of different values of m and n, and only activate the heuristic at query time if the queried group size falls in a pre-test range where the heuristic shows better performance.
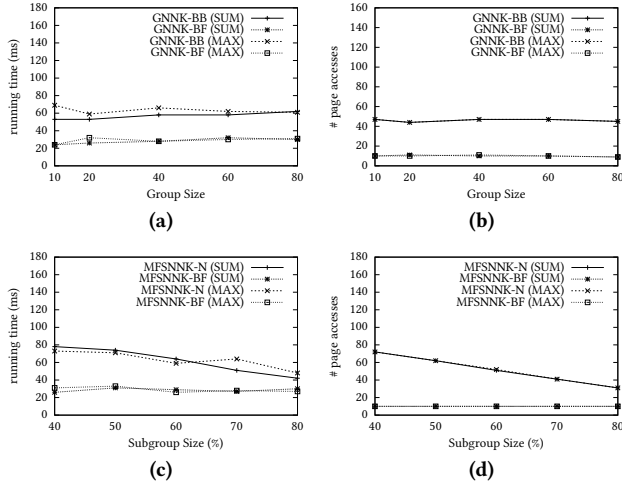

Figure 6: The effect of varying query group size (a-b) and minimum subgroup size (c-d) in running time and I/O

6.5 Experiments on Yelp dataset

We have run the same set of experiments as mentioned above on the Yelp dataset. All of our experimental results show similar trends in both datasets. Due to page limitations, we only present the experimental results for varying group size for GNNK queries and minimum subgroup size for MFSNNK queries with Yelp dataset in Figure 6 (a-b) and Figure 6 (c-d), respectively.

7 CONCLUSION

We presented a new type of group spatial keyword query suitable for a collaborative environment. This query aims to find the best POI that minimizes the aggregate distance and maximizes the text relevancy for a group of users. We have studied three instances of this query, which return (i) the best POI for the whole group, (ii) the optimal subgroup with the best POI given a subgroup size m, and (iii) the optimal subgroups and the corresponding best POIs of different subgroup sizes in $m, m + 1, ..., n$. In all these queries,

our proposed best-first approach runs approximately 4 times faster (on average) than the branch and bound approach for both real datasets.

This study brings a number of future studies. For example, a study that allows users to set the value of α to reflect their preference of spatial proximity over textual relevance would make the query more user friendly. Also, extending the algorithms to road networks would further improve their practicality.

REFERENCES

[1] Mohammed Eunus Ali, Egemen Tanin, Peter Scheuermann, Sarana Nutanong, and Lars Kulik. 2016. Spatial Consensus Queries in a Collaborative Environment. *TSAS* 2, 1 (2016), 3:1–3:37.
[2] Stefan Berchtold, Christian Böhm, Daniel A Keim, and Hans-Peter Kriegel. 1997. A cost model for nearest neighbor search in high-dimensional data space. In *PODS*. 78–86.
[3] Andrei Z Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *CIKM*. 426–434.
[4] Xin Cao, Gao Cong, Tao Guo, Christian S Jensen, and Beng Chin Ooi. 2015. Efficient Processing of Spatial Group Keyword Queries. *TODS* 40, 2 (2015), 13.
[5] Xin Cao, Gao Cong, and Christian S Jensen. 2010. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB* 3, 1-2 (2010), 373–384.
[6] Xin Cao, Gao Cong, Christian S Jensen, and Beng Chin Ooi. 2011. Collective spatial keyword querying. In *SIGMOD*. 373–384.
[7] Kaushik Chakrabarti, Surajit Chaudhuri, and Venkatesh Ganti. 2011. Interval-based pruning for top-k processing over compressed lists. In *ICDE*. 709–720.
[8] Kunjie Chen, Weiwei Sun, Chuanchuan Tu, Chunan Chen, and Yan Huang. 2012. Aggregate keyword routing in spatial database. In *GIS*. 430–433.
[9] Gao Cong, Christian S Jensen, and Dingming Wu. 2009. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB* 2, 1 (2009), 337–348.
[10] Shuai Ding and Torsten Suel. 2011. Faster top-k document retrieval using block-max indexes. In *SIGIR*. 993–1002.
[11] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. System Sci.* 66, 4 (2003), 614–656.
[12] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*. 47–57.
[13] Gisli R Hjaltason and Hanan Samet. 1995. Ranking in spatial databases. In *SSD*. 83–95.
[14] Gísli R Hjaltason and Hanan Samet. 1999. Distance browsing in spatial databases. *TODS* 24, 2 (1999), 265–318.
[15] Yang Li, Feifei Li, Ke Yi, Bin Yao, and Min Wang. 2011. Flexible aggregate similarity search. In *SIGMOD*. 1009–1020.
[16] Zhisheng Li, Ken CK Lee, Baihua Zheng, Wang-Chien Lee, Dik Lee, and Xufa Wang. 2011. Ir-tree: An efficient index for geographic document search. *TKDE* 23, 4 (2011), 585–599.
[17] Ying Lu, Jiaheng Lu, Gao Cong, Wei Wu, and Cyrus Shahabi. 2014. Efficient algorithms and cost models for reverse spatial-keyword k-nearest neighbor search. *ACM Transactions on Database Systems (TODS)* 39, 2 (2014), 13.
[18] Dimitris Papadias, Qiongmao Shen, Yufei Tao, and Kyriakos Mouratidis. 2004. Group nearest neighbor queries. In *ICDE*. 301–312.
[19] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. 2005. Aggregate nearest neighbor queries in spatial databases. *TODS* 30, 2 (2005), 529–576.
[20] Jay M Ponte and W Bruce Croft. 1998. A language modeling approach to information retrieval. In *SIGIR*. 275–281.
[21] João B Rocha-Junior, Orestis Gkorgkas, Simon Jonassen, and Kjetil Nørvåg. 2011. Efficient processing of top-k spatial keyword queries. In *SSTD*. 205–222.
[22] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. 1995. Nearest neighbor queries. In *ACM SIGMOD Record*, Vol. 24. 71–79.
[23] Kai Yao, Jianjun Li, Guohui Li, and Changyin Luo. 2016. Efficient Group Top-k Spatial Keyword Query Processing. In *ApWeb*. 153–165.
[24] Dongxiang Zhang, Yeow Meng Chee, Anirban Mondal, Anthony KH Tung, and Masaru Kitsuregawa. 2009. Keyword search in spatial databases: Towards searching by document. In *ICDE*. 688–699.
[25] Yuxin Zheng, Zhifeng Bao, Lidan Shou, and Anthony K. H. Tung. 2015. INSPIRE: A Framework for Incremental Spatial Prefix Query Relaxation. *TKDE* 27, 7 (2015), 1949–1963.