Top-k Spatial Preference Queries*

Man Lung Yiu
Department of Computer Science
Aalborg University
DK-9220 Aalborg, Denmark
mly@cs.aau.dk

Xiangyuan Dai, Nikos Mamoulis
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
{xydai, nikos}@cs.hku.hk

Michail Vaitis
Department of Geography
University of the Aegean
Mytilene, Greece
vaitis@aegean.gr

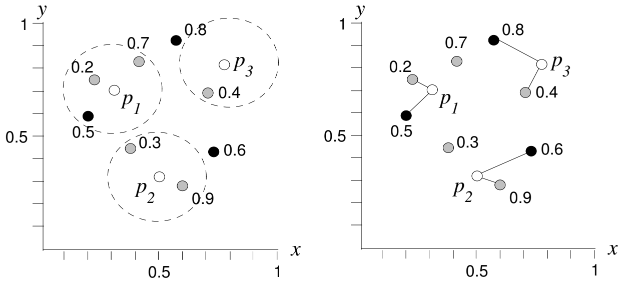
Abstract

A spatial preference query ranks objects based on the qualities of features in their spatial neighborhood. For example, consider a real estate agency office that holds a database with available flats for lease. A customer may want to rank the flats with respect to the appropriateness of their location, defined after aggregating the qualities of other features (e.g., restaurants, cafes, hospital, market, etc.) within a distance range from them. In this paper, we formally define spatial preference queries and propose appropriate indexing techniques and search algorithms for them. Our methods are experimentally evaluated for a wide range of problem settings.

1 Introduction

Spatial database systems manage large collections of geographic entities, which apart from spatial attributes contain non-spatial information (e.g., name, size, type, price, etc.). In this paper, we study an interesting type of preference queries, which select the best spatial location with respect to the quality of facilities in its spatial neighborhood.

Given a set $\mathcal{D}$ of interesting *objects* (e.g., candidate locations), a top-k *spatial preference query* retrieves the k objects in $\mathcal{D}$ with the highest scores. The score of an object is defined by the quality of *features* (e.g., facilities or services) in its spatial neighborhood. As a motivating example, consider a real estate agency office that holds a database with available flats for lease. A customer may want to rank the contents of this database with respect to the quality of their locations, quantized by aggregating non-spatial characteristics of other features[1] (e.g., restaurants, cafes, hospital, market, etc.) in the spatial neighborhood of the flat (defined by a spatial range around it). Quality may be subjective and query-parametric. For example, a user may define quality with respect to non-spatial attributes of restaurants around it (e.g., whether they serve seafood, price range, etc.).



(a) Range score (within 0.2 km) (b) Nearest neighbor score

Figure 1. Examples of top-k spatial preference queries

The white points of Figure 1a are an exemplary object dataset $\mathcal{D}$ (e.g., hotels). In the figure, there are also two feature datasets; the gray points of $\mathcal{F}_1$ (restaurants) and the black points of $\mathcal{F}_2$ (cafes). Feature points are labeled by non-spatial (quality) values (e.g., available from rating providers, like http://www.zagat.com/). A tourist may be interested in the hotel p that maximizes a score $\tau(p)$, defined as the sum of maximum restaurant quality and maximum cafe quality in the neighborhood of p (e.g, dotted circle at p with radius of 0.2 km). For instance, the maximum quality of gray and black points within the circle of p_1 are 0.7 and 0.5 respectively, so the score of p_1 is $\tau(p_1) = 0.7 + 0.5 = 1.2$. Since there are no black points within the range of p_2, we have $\tau(p_2) = 0.9 + 0 = 0.9$. Similarly, we obtain $\tau(p_3) = 0.4 + 0 = 0.4$. Hence, the hotel p_1 is returned as the top result to the user. As an alternative example, Figure 1b illustrates the case where the score $\tau(p)$ of a hotel is taken as the sum of qualities of its *nearest* restaurant and cafe (indicated by connecting line segments). Hence, we have $\tau(p_1) = 0.2 + 0.5 = 0.7$, $\tau(p_2) = 0.9 + 0.6 = 1.5$, and $\tau(p_3) = 0.4 + 0.8 = 1.2$. The best hotel in this case is p_2.

Traditionally, there are two basic ways for ranking objects: (i) spatial ranking, which orders the objects according to their distance from a reference point, and (ii) non-spatial

*Work supported by grant 7160/05E from Hong Kong RGC.

[1] Here "feature" denotes a class of objects in a spatial map such as specific facilities or services.

ranking, which orders the objects by an aggregate function on their non-spatial values. Our top-k spatial preference query integrates these two types of ranking in an intuitive way. As indicated by our examples, this new query has a wide range of applications in service recommendation and decision support systems.

Despite the usefulness of the top-k spatial preference query, to our knowledge, it has not been studied in the past. A brute-force approach (to be elaborated in Section 3.1) for evaluating it is to compute the scores of all objects in $\mathcal{D}$ and select the top-k ones. This method, however, is expected to be very expensive for large input datasets. In this paper, we propose alternative techniques that aim at minimizing the I/O accesses to the object and feature datasets, while being also computationally efficient. Our techniques apply on spatial-partitioning access methods and compute upper score bounds for the objects indexed by them, which are used to effectively prune the search space.

The rest of this paper is structured as follows. In Section 2 we provide background on basic and advanced queries on spatial databases, as well as top-k query evaluation in relational databases. Section 3 models the queries that we study in this paper and presents the suggested solutions. In Section 4, our query algorithms are experimentally evaluated with real and synthetic data. Section 5 discusses some extensions of the problem. Finally, Section 6 concludes the paper with future research directions.

2 Background and Related Work

Object ranking is a popular retrieval task in various applications. In relational databases, we often want to rank tuples using an aggregate score function on their attribute values [5]. For example, consider a database of a real estate agency, containing information about flats available for rent. A potential customer may want to view the top-10 flats with the largest sizes and lowest prices. The score of each flat in this case is expressed by the sum of two individual scores: size and price, after they have been scaled to the same range (e.g., between 0 and 1, where 1 indicates the highest preference; highest possible size and lowest possible price).

Another popular object ranking application is document ranking based on the relevance of the keywords (terms) they contain to a user query (also expressed by a set of terms). This problem has been the primary research in information retrieval (IR) for over two decades [1]. The ranking function in this problem is again an aggregation of the relevance of the query terms with the document, often enriched with some global ranking scores of the documents according to their popularity [4].

In spatial databases, ranking is often associated to nearest neighbor (NN) retrieval. Given a query location, we are often interested in retrieving the set of nearest objects to it that qualify a condition (e.g., restaurants). Assuming that the set of interesting objects is indexed by a hierarchical spatial access method (e.g., the R-tree [9]), we can use distance bounds while traversing the index to derive the answer in a branch-and-bound fashion [10]. Tao et al. [17] noted that top-k queries can be modeled as (weighted) nearest neighbor queries, in the multi-dimensional space defined by the involved attribute domains, where the query point is formed by taking the maximum value of each dimension. Motivated by this observation, they adapted the algorithm of [10] for this problem.

Nevertheless, it is not always possible to use multidimensional indexes for top-k retrieval. First, such indexes usually break-down in high dimensional spaces [18, 3]. Second, top-k queries may involve an arbitrary set of attributes (e.g., size and price) from a set of possible ones (e.g., size, price, distance to the beach, number of bedrooms, floor, etc.) and indexes may not be available for all possible attribute combinations (i.e., they are too expensive to create and maintain). Third, information for different rankings to be combined (i.e., for different attributes) could appear in different databases (in a distributed database scenario) and unified indexes may not exist for them. A stream of research [8, 5, 11, 12] for top-k queries has focused on the efficient merging of object rankings that may arrive from different (distributed) sources. The motivation of these methods is to minimize the number of accesses to the input rankings until the objects with the top-k aggregate scores have been identified. To achieve this, upper and lower bounds for the objects seen so far are maintained while traversing the sorted lists.

In the next paragraphs, we first review the R-tree, which is the most popular spatial access method and the NN search algorithm of [10] and survey recent research of feature-based spatial queries.

2.1 Spatial Query Evaluation on R-trees

The most popular spatial access method is the R-tree [9], which indexes minimum bounding rectangles (MBRs) of objects. Figure 2 shows a collection $R = \{p_1, \ldots, p_8\}$ of spatial objects (e.g., points) and an R-tree structure that indexes them.

R-trees can efficiently process main spatial query types, including spatial range queries, nearest neighbor queries, and spatial joins. Given a spatial region W, a *spatial range query* retrieves from R the objects that intersect W. For instance, consider a range query that asks for all objects within distance 3 from q, corresponding to the shaded area in Figure 2. Starting from the root of the tree, the query is processed by recursively following entries, having MBRs that intersect the query region. For instance, e_1 does not intersect the query region, thus the subtree pointed by e_1 cannot contain any query result. In contrast, e_2 is followed by the search algorithm and the points in the corresponding node are examined recursively to find the query result p_7.

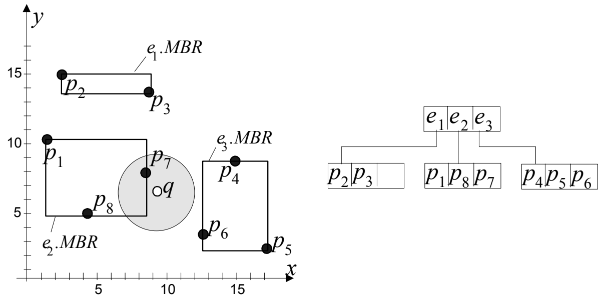


Figure 2. Spatial queries on R-trees

A nearest neighbor (NN) query takes as input a query object q and returns the closest object in R to q. For instance, the nearest neighbor of q in Figure 2 is p_7. A popular generalization is the k-NN query, which returns the k closest objects to q, given a positive integer k. NN (and k-NN) queries can be efficiently processed using the *best-first* (BF) algorithm of [10], provided that R is indexed by an R-tree. A priority queue PQ which organizes R-tree entries based on the (minimum) distance of their MBRs to q is initialized with the root entries. In order to find the NN of q in Figure 2, BF first inserts to PQ entries e_1, e_2, e_3, and their distances to q. Then the nearest entry e_2 is retrieved from PQ and objects p_1, p_7, p_8 are inserted to PQ. The next nearest entry in PQ is p_7, which is the nearest neighbor of q. In terms of I/O, the BF algorithm is shown to be no worse than any NN algorithm that applies on the same R-tree [10]. A less efficient approach is the *depth first* (DF) algorithm [16], which traverses the tree in the depth-first fashion. As shown in [10], DF can be more I/O consuming than BF; however, DF requires only bounded memory and at most a single tree path resides in memory during search.

The *aggregate* R-tree (aR-tree) [15] is a variant of the R-tree, where each non-leaf entry augments an aggregate measure for some attribute value (measure) of all points in its subtree. As an example, the tree shown in Figure 2 can be upgraded to a MAX aR-tree over the point set, if entries e_1, e_2, e_3 contain the maximum measure values of sets $\{p_2, p_3\}, \{p_1, p_8, p_7\}, \{p_4, p_5, p_6\}$, respectively. Assume that the measure values of p_4, p_5, p_6 are $0.2, 0.1, 0.4$, respectively. In this case, the aggregate measure augmented in e_3 would be $\max\{0.2, 0.1, 0.4\} = 0.4$. In Section 3, we will illustrate how MAX aR-trees, indexing the feature datasets (e.g., restaurants), can be used to accelerate the processing of top-k spatial preference queries.

2.2 Feature-based Spatial Queries

Recently, [19] solved the problem of finding top-k sites (e.g., restaurants) based on their *influence* on feature points (e.g., residential buildings). As an example, Figure 3 shows a set of sites (white points), a set of features (black points with weights), such that each dotted line links a feature point to its nearest site. The influence of a site p_i is defined by the sum of weights of feature points having p_i as their closest site. For instance, the score of p_1 is 0.9+0.5=1.4. Similarly, the scores of p_2 and p_3 are 1.5 and 1.2 respectively. Hence, p_2 is returned as the top-1 influential site.

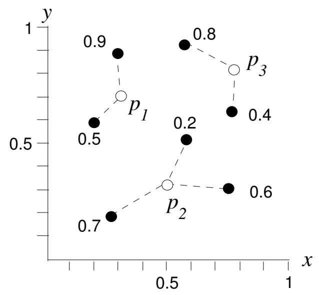
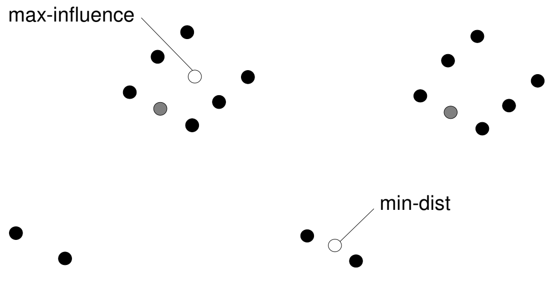


Figure 3. Top-k influential sites

Related to top-k influential sites query are the optimal location queries studied in [7, 20]. The goal is to find the location in space (not chosen from a specific set of sites) that minimizes an objective function. In Figures 4a and 4b, feature points and existing sites are shown as black and gray points respectively. Assume that all feature points have the same quality. The maximum influence optimal location query [7] finds the location (to insert to the existing set of sites) with the maximum influence (as defined in [19]), whereas the minimum distance optimal location query [20] searches for the location that minimizes the average distance from each feature point to its nearest site. The optimal locations for both queries are marked as white points in Figures 4a,b respectively.



(a) Max-influence (b) Min-distance

Figure 4. Optimal location queries

The techniques proposed in [19, 7, 20] are specific to the particular query types described above and cannot be extended for the class of top-k spatial preference queries, which we study in this paper. Also, they deal with a single feature dataset whereas our queries consider multiple feature datasets. Another piece of related work to ours is a web-search engine for evaluating textual geographic queries by considering also the spatial context of the searched documents [6].

3 Algorithms for Spatial Preference Queries

A top-k spatial preference query retrieves the k points in an *object* dataset $\mathcal{D}$ (i.e., set of interesting points) with the highest score. Given an object point $p \in \mathcal{D}$ and m *feature* datasets $\mathcal{F}_1, \cdots, \mathcal{F}_m$, the score of p is defined as

$$\tau^{\theta}(p) = agg\{\ \tau_c^{\theta}(p) \mid c \in [1, m]\ \} \tag{1}$$

where agg is a monotone aggregate operator and $\tau_c^{\theta}(p)$ is the (c-th) component score of p with respect to the neighborhood condition θ and the (c-th) feature dataset $\mathcal{F}_c$. Typical examples of the aggregate function agg are: `SUM`, `MIN`, `MAX`. In practice, θ models the spatial neighborhood region of a point $p \in \mathcal{D}$. Two intuitive choices for the component score function $\tau_c^{\theta}(p)$ are:

- the range score[2] $\tau_c^{rng}(p)$, taken as the maximum *quality* $\omega(s)$ of points $s \in \mathcal{F}_c$ that are within a given parameter distance ϵ_c from p, or 0 if no such point exists.
- the nearest neighbor (NN) score $\tau_c^{nn}(p)$, taken as the *quality* $\omega(s)$ of s; the NN of p in $\mathcal{F}_c$.

The quality $\omega(s)$ of a feature object (e.g., restaurant) may correspond to a known value from a ratings provider. Although in this paper we study $\tau_c^{rng}(p)$ and $\tau_c^{nn}(p)$ scoring functions as test cases, other user-defined aggregate functions can be integrated into our framework.

For simplicity, we assume that the quality $\omega(s)$ of a feature point s lies within the interval $[0, 1]$. Thus, for a point $p \in \mathcal{D}$, where not all its component scores are known, its upper bound score $\tau_+^{\theta}(p)$ is defined as:

$$\tau_+^{\theta}(p) = agg\ _{c=1}^{m} \begin{cases} \tau_c^{\theta}(p) & \text{if } \tau_c^{\theta}(p) \text{ is known} \\ 1 & \text{otherwise} \end{cases} \tag{2}$$

Note that, for any monotone aggregate operator agg, it is guaranteed that the bound $\tau_+^{\theta}(p)$ is greater than or equal to the actual score $\tau^{\theta}(p)$.

In the remainder of the paper, we assume that the object dataset $\mathcal{D}$ is indexed by an R-tree and each feature dataset $\mathcal{F}_c$ is indexed by an `MAX` aR-tree, where each non-leaf entry augments the maximum quality (of features) in its subtree. Nevertheless, our solutions are generic and can be adapted for the cases where the datasets are indexed by other hierarchical spatial indexes (e.g., point quad-trees). The rationale of indexing different feature datasets by separate aR-trees is that: (i) a user queries for only few features (e.g., restaurants and cafes) out of all possible features (e.g., restaurants, cafes, hospital, market, etc.), and (ii) different users may consider different subsets of features.

Based on the above indexing scheme, we develop various algorithms for processing top-k spatial preference queries. For the ease of discussion, we assume the aggregate operator agg to be `SUM`, although our techniques can directly be applied for arbitrary monotone aggregate functions. Whenever the context is clear, the condition θ is dropped.

3.1 Probing Algorithms

We first introduce a brute-force solution that computes the score of every point $p \in \mathcal{D}$ in order to obtain the query results. Then, we propose a group evaluation technique that computes the scores of multiple points concurrently.

Algorithm 1 Simple Probing Algorithm (SP)

```
    algorithm SP(Node $N$)
 1: for all entries $e \in N$ do
 2:     if $N$ is non-leaf then
 3:         read the child node $N'$ pointed by $e$;
 4:         SP($N'$);
 5:     else
 6:         for $c$:=1 to $m$ do
 7:             if $\tau_+(e) > \gamma$ then          ▷ upper bound score
 8:                 compute $\tau_c(e)$ by using the tree $\mathcal{F}_c$;
 9:         if $\tau(e) > \gamma$ then
10:             update $W_k$ (and $\gamma$) by $e$;
```

Algorithm 1 is a pseudo-code of the simple probing algorithm (SP), which retrieves the query results by computing the score of every object point. The algorithm uses two global variables: W_k is a min-heap for managing the top-k results and γ represents the top-k score so far (i.e., lowest score in W_k). Initially, the algorithm is invoked at the root node of the object tree (i.e., $N = \mathcal{D}.root$). The procedure is recursively applied (at Line 4) on tree nodes until a leaf node is accessed. When a leaf node is reached, the component score $\tau_c(e)$ (at Line 8) is computed by executing a range search (NN search) on the feature tree $\mathcal{F}_c$ for range score (NN score) queries. Lines 6-8 describe an *incremental computation* technique, for reducing unnecessary component score computations. In particular, the point e is ignored as soon as its upper bound score $\tau_+(e)$ (see Equation 2) cannot be greater than the best-k score γ. On the other hand, W_k and γ are updated when the actual score $\tau(e)$ is greater than γ.

Group Probing Algorithm Due to separate score computations for different objects, SP is inefficient for large object datasets. In view of this, we propose the group probing algorithm (GP), a variant of SP, that reduces I/O cost by computing scores of objects in the same leaf node of the R-tree concurrently. In GP, when a leaf node is visited, its points are first stored in a set V and then their component scores are computed concurrently at a single traversal of the $\mathcal{F}_c$ tree.

Algorithm 2 shows the procedure for computing the c-th component score for a group of points. Consider a subset V of $\mathcal{D}$ for which we want to compute their $\tau_c^{rng}(p)$

[2] The value ϵ_c is an implicit parameter for the range score $\tau_c^{rng}(p)$.

score at feature tree $\mathcal{F}_c$. Initially, the procedure is called with N being the root node of $\mathcal{F}_c$. If e is a non-leaf entry and its $mindist$[3] from some point $p \in V$ is within the range ϵ_c, then the procedure is applied recursively on the child node of e, since the sub-tree of $\mathcal{F}_c$ rooted at e may contribute to the component score of p. In case e is a leaf entry (i.e., a feature point), the scores of points in V are updated if they are within distance ϵ_c from e. An additional optimization (for range score computation only) is to replace the condition at Line 3 by a stronger one: "$\exists p \in V,\ mindist(p, e) \le \epsilon_c \wedge \omega(e) > \tau_c(p)$", in which the quality $\omega(e)$ is stored in the current aR-tree node N. In other words, even when e may contain feature points within distance ϵ_c from p, if it cannot improve the component score of p, then its child node N' needs not be visited.

Algorithm 2 Group Range Score Algorithm

```
    algorithm Group_Range(Node $N$, Set $V$, Value $c$, $\epsilon_c$)
1:  for all entry $e \in N$ do
2:      if $N$ is non-leaf then
3:          if $\exists p \in V,\ mindist(p, e) \le \epsilon_c$ then
4:              read the child node $N'$ pointed by $e$;
5:              Group_Range($N'$,$V$,$c$,$\epsilon_c$);
6:      else
7:          for all $p \in V$ such that $dist(p, e) \le \epsilon_c$ do
8:              $\tau_c(p)$:=max$\{\tau_c(p), \omega(e)\}$;
```

We now discuss how Algorithm 2 can be modified for computing NN scores. Instead of using a single range ϵ_c, we associate each point $p \in V$ with $p.\epsilon_c$; its nearest distance to feature points in $\mathcal{F}_c$ found so far ($p.\epsilon_c$ is initialized to ∞). In addition, at Lines 7-8, we check whether e replaces the NN of a $p \in V$ and if so, we set $\tau_c(p) := \omega(e)$ and $p.\epsilon_c := dist(p, e)$. Before Line 1, we sort the entries $e \in N$ in ascending order of $mindist(\overline{V}, e)$, where $\overline{V}$ is the centroid of all points in V. In this way, entries closer to points in V are visited earlier and fewer tree nodes in $\mathcal{F}_c$ are accessed. Note that the above algorithm traverses the tree in a depth-first manner [16]. Its I/O cost can be reduced by traversing the tree in a best-first manner [10]. For this, a global priority queue PQ is used for organizing visited tree entries in ascending order of their distance from $\overline{V}$.

3.2 Branch and Bound Algorithm

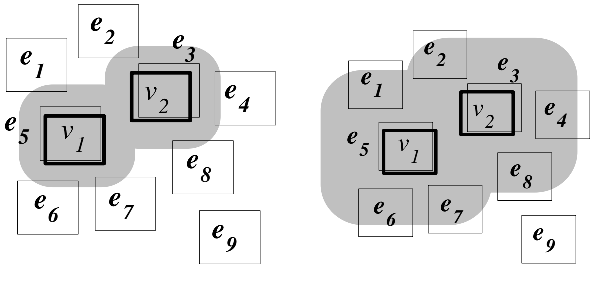
GP is still expensive as it examines all objects in $\mathcal{D}$ and computes their component scores. We now propose an algorithm that can significantly reduce the number of objects to be examined. The key idea is to compute, for non-leaf entries e in the object tree $\mathcal{D}$, an upper bound $\mathcal{T}(e)$ (but not $\tau(e)$) of the score for any point in the subtree of e. If $\mathcal{T}(e) \le \gamma$, then we need not access the subtree of e (and also save numerous score computations).

Algorithm 3 is a pseudo-code of our branch and bound algorithm (BB), based on this idea. BB is called with N being the root node of $\mathcal{D}$. If N is a non-leaf node, Lines 3-5 compute the scores $\mathcal{T}(e)$ for non-leaf entries e concurrently. Recall that $\mathcal{T}(e)$ is an upper bound score for any point in e. The techniques for computing $\mathcal{T}(e)$ will be discussed shortly. Like Equation 2, with the component scores $\mathcal{T}_c(e)$ known so far, we can derive $\mathcal{T}_+(e)$, an upper bound of $\mathcal{T}(e)$. If $\mathcal{T}_+(e) \le \gamma$, then the subtree of e cannot contain better results than those in W_k and it is removed from V. In order to obtain points with high scores early, we sort the entries in descending order of $\mathcal{T}(e)$ before invoking the above procedure recursively on the child nodes pointed by the entries in V. If N is a leaf node, we compute the scores for all points of N concurrently and then update the set W_k of the top-k results. Since both W_k and γ are global variables, the value of γ is updated during recursive call of BB.

Algorithm 3 Branch and Bound Algorithm (BB)

```
    $W_k$:=new min-heap of size $k$ (initially empty);
    $\gamma$:=0;                                   $\triangleright$ $k$-th score in $W_k$

    algorithm BB(Node $N$)
1:  $V$:=$\{e|e \in N\}$;
2:  if $N$ is non-leaf then
3:      for $c$:=1 to $m$ do
4:          compute $\mathcal{T}_c(e)$ for all $e \in V$ concurrently;
5:          remove entries $e$ in $V$ such that $\mathcal{T}_+(e) \le \gamma$;
6:      sort entries $e \in V$ in descending order of $\mathcal{T}(e)$;
7:      for all entry $e \in V$ such that $\mathcal{T}(e) > \gamma$ do
8:          read the child node $N'$ pointed by $e$;
9:          BB($N'$);
10: else
11:     for $c$:=1 to $m$ do
12:         compute $\tau_c(e)$ for all $e \in V$ concurrently;
13:         remove entries $e$ in $V$ such that $\tau_+(e) \le \gamma$;
14:     update $W_k$ (and $\gamma$) by entries in $V$;
```



(a) Range score (b) Nearest neighbor score

Figure 5. Deriving upper bound scores

It remains to clarify how the (upper bound) scores of non-leaf entries (within the same node N) can be computed

[3] Given a point p and an MBR e, $mindist(p, e)$ ($maxdist(p, e)$) [10] denotes the minimum (maximum) possible distance between p and a point in e. Similarly, $mindist(e_a, e_b)$ ($maxdist(e_a, e_b)$) denotes the minimum (maximum) possible distance between a point in MBR e_a and a point in MBR e_b.

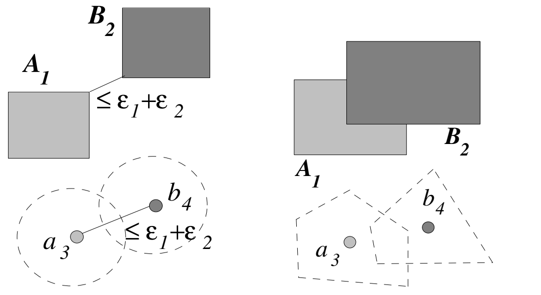
concurrently (at Line 4). Our goal is to compute these upper bound scores with low I/O cost and the bounds not to be too loose, in order for pruning to be effective. For this, we utilize only level-1 entries (i.e., lowest level non-leaf entries) in $\mathcal{F}_c$ for deriving upper bound scores because: (i) there are much fewer level-1 entries than leaf entries, and (ii) high level entries in $\mathcal{F}_c$ cannot provide tight bounds. Algorithm 2 can be used for this purpose (where input V corresponds to a set of non-leaf entries), after changing Line 2 to check whether child nodes of N are above the leaf level.

The following example illustrates how upper bound range scores are derived. In Figure 5a, v_1 and v_2 are non-leaf entries in the object tree $\mathcal{D}$ and the others are level-1 entries in the feature tree $\mathcal{F}_c$. For the entry v_1, we first define its Minkowski region [2] (i.e., gray region around v_1), the area whose $mindist$ from v_1 is within ϵ_c. Observe that only entries e_i intersecting the Minkowski region of v_1 can contribute to the score of some point in v_1. Thus, the upper bound score $\mathcal{T}_c(v_1)$ is simply the maximum quality of entries e_1, e_5, e_6, e_7. Similarly, $\mathcal{T}_c(v_2)$ is computed as the maximum quality of entries e_2, e_3, e_4, e_8. Assuming that v_1 and v_2 are entries in the same tree node of $\mathcal{D}$, their upper bounds are computed concurrently to reduce I/O cost.

Upper bound NN scores for the entries v_1, v_2 of Figure 5b can be derived as follows. For v_1, we first find the level-1 entry in $\mathcal{F}_c$ with the smallest $maxdist$ from v_1. This corresponds to entry e_5. It is guaranteed that the NN of any $p \in v_1$ must intersect the Minkowski region covering the area whose $mindist$ from v_1 is within $v_1.\epsilon_c = maxdist(v_1, e_5)$. Thus, the upper bound $\mathcal{T}_c(v_1)$ is taken as the maximum quality of the entries $e_1, e_2, e_3, e_5, e_6, e_7$. In fact, the two steps above are combined to a single traversal of the tree on $\mathcal{F}_c$. Again, group computation is performed for non-leaf entries (e.g., v_1 and v_2) located in the same node of the object tree $\mathcal{D}$, as follows: (i) for each level-1 entry encountered (in the tree on $\mathcal{F}_c$), update each $v_i.\epsilon_c$ and insert the entry into a list Φ, (ii) prune an entry if it does not fall in the Minkowski region for all v_i, and (iii) after the tree traversal, derive the upper bound for each v_i from the (relevant) entries in Φ.

3.3 Feature Join Algorithm

An alternative method for evaluating a top-k spatial preference query is to perform a multi-way spatial join [13] on the feature trees $\mathcal{F}_1, \mathcal{F}_2, \cdots, \mathcal{F}_m$ to obtain combinations of feature points which can be in the neighborhood of some object from $\mathcal{D}$. Spatial regions which correspond to combinations of high scores are then examined, in order to find data objects in $\mathcal{D}$ having the corresponding feature combination in their neighborhood. In this section, we first introduce the concept of a combination, then discuss the conditions for a combination to be pruned, and finally elaborate the algorithm used to progressively identify the combinations that correspond to query results.



(a) Range score (b) Nearest neighbor score

Figure 6. Qualified combinations for the join

Tuple $\langle f_1, f_2, \cdots, f_m \rangle$ is a *combination* if, for any $c \in [1, m]$, f_c is an entry (either leaf or non-leaf) in the feature tree $\mathcal{F}_c$. The score of the combination is defined by:

$$\tau(\langle f_1, f_2, \cdots, f_m \rangle) = agg \, {}_{c=1}^{m} \, \omega(f_c) \tag{3}$$

For a non-leaf entry f_c, $\omega(f_c)$ is the MAX of all feature qualities in its subtree (stored with f_c, since $\mathcal{F}_c$ is an aR-tree). A combination disqualifies a range score query if:

$$\exists \, (i \neq j \wedge i, j \in [1, m]), \; mindist(f_i, f_j) > \epsilon_i + \epsilon_j \tag{4}$$

When such a condition holds, it is impossible to have a point in $\mathcal{D}$ whose $mindist$ from f_i and f_j are within ϵ_i and ϵ_j respectively. The above validity check acts as a multiway join condition that significantly reduces the number of combinations to be examined.

Figure 6a shows graphically the condition for a pair of entries (e.g., non-leaf combination $\langle A_1, B_2 \rangle$, leaf combination $\langle a_3, b_4 \rangle$) to be a candidate combination for a range preference query. Figure 6b illustrates the corresponding condition for the case of NN preference queries. The dotted polygon around the point a_3 (of $\mathcal{F}_1$) corresponds to its Voronoi cell [14] in $\mathcal{F}_1$; it is guaranteed that any point $p \in \mathcal{D}$ located in the Voronoi cell of a_3 must have a_3 as its nearest feature point in $\mathcal{F}_1$. The same holds for other points (e.g., the feature point b_4 of $\mathcal{F}_2$).

We can use feature trees $\mathcal{F}_c$ that index the Voronoi cells (together with the feature points), to accelerate processing of NN score queries. In specific, a combination $\langle f_1, f_2, \cdots, f_m \rangle$ disqualifies such a query if:

$$\bigcap_{c \in [1,m]} f_c = \varnothing \tag{5}$$

Algorithm 4 is a pseudo-code of our feature join (FJ) algorithm (used for range and NN preference queries). The algorithm employs a max-heap H for managing combinations of feature entries in descending order of their combination scores. The score of a combination $\langle f_1, f_2, \cdots, f_m \rangle$

Algorithm 4 Feature Join Algorithm (FJ)

```
    $W_k$:=new min-heap of size $k$ (initially empty);
    $\gamma$:=0;                                          ▷ $k$-th score in $W_k$

    algorithm FJ(Tree $\mathcal{D}$,Trees $\mathcal{F}_1, \mathcal{F}_2, \cdots, \mathcal{F}_m$)
1:  $H$:=new max-heap (combination score as the key);
2:  insert $\langle \mathcal{F}_1.root, \mathcal{F}_2.root, \cdots, \mathcal{F}_m.root \rangle$ into $H$;
3:  while $H$ is not empty do
4:      deheap $\langle f_1, f_2, \cdots, f_m \rangle$ from $H$;
5:      if $\forall\ c \in [1, m],\ f_c$ points to a leaf node then
6:          for $c$:=1 to $m$ do
7:              read the child node $L_c$ pointed by $f_c$;
8:          Find_Result($\mathcal{D}.root$, $L_1, \cdots, L_m$);
9:      else
10:         $f_c$:=highest level entry among $f_1, f_2, \cdots, f_m$;
11:         read the child node $N_c$ pointed by $f_c$;
12:         for all entry $e_c \in N_c$ do
13:             insert $\langle f_1, f_2, \cdots, e_c, \cdots, f_m \rangle$ into $H$ if its
    score is greater than $\gamma$ and it qualifies the query;

    algorithm Find_Result(Node $N$, Nodes $L_1, \cdots, L_m$)
1:  for all entries $e \in N$ do
2:      if $N$ is non-leaf then
3:          compute $\mathcal{T}(e)$ by entries in $L_1, \cdots, L_m$;
4:          if $\mathcal{T}(e) > \gamma$ then
5:              read the child node $N'$ pointed by $e$;
6:              Find_Result($N'$, $L_1, \cdots, L_m$);
7:      else
8:          compute $\tau(e)$ by entries in $L_1, \cdots, L_m$;
9:          update $W_k$ (and $\gamma$) by $e$ (when necessary);
```

as defined in Equation 3 is an upper bound of the scores of all combinations $\langle s_1, s_2, \cdots, s_m \rangle$ of feature points, such that s_c is located in the subtree of f_c for each $c \in [1, m]$. Initially, the combination with the root pointers of all feature trees is enheaped. We progressively deheap the combination with the largest score. If all its entries point to leaf nodes, then we load these nodes $L_1, \cdots, L_m$ and call Find_Result to traverse the object R-tree $\mathcal{D}$ and find potential results. Find_Result is a variant of the BB algorithm, with the following differences: (i) $L_1, \cdots, L_m$ are viewed as m tiny feature trees (each with one node) and accesses to them incur no extra I/O cost, and (ii) for the case of NN score, if the entry e does not intersect any qualified combination formed by feature entries in $L_1, \cdots, L_m$, then $\mathcal{T}(e)$ (at Line 3) and $\tau(e)$ (at Line 8) are set to 0.

In case not all entries of the deheaped combination point to leaf nodes (Line 9 of FJ), we select the one at the highest level, access its child node N_c and then form new combinations with the entries in N_c. A new combination is inserted into H for further processing if its score is higher than γ and it qualifies the query. The loop (at Line 3) continues until H becomes empty.

4 Experimental Evaluation

In this section, we compare the efficiency of the proposed algorithms using real and synthetic datasets. Each dataset is indexed by an aR-tree with 4K bytes page size. We used an LRU memory buffer whose default size is set to 0.5% of the sum of tree sizes (for the object and feature trees used). Our algorithms were implemented in C++ and experiments were run on a Pentium IV 2.3GHz PC with 512 MB of RAM. In all experiments, we measure only the I/O cost (i.e., number of page faults) of the algorithms as their CPU costs follow similar trends. Section 4.1 describes the experimental settings, while Section 4.2 presents our experimental findings.

4.1 Experimental Settings

We used both real and synthetic data for the experiments. For each synthetic dataset, the coordinates of points are random values uniformly and independently generated for different dimensions. By default, an object dataset contains 200K points and a feature dataset contains 100K points. The real datasets are described in Table 1. All of them are geographical datasets of China, available at the *Digital Chart of the World*, http://www.maproom.psu.edu/dcw/. The point coordinates of all (real and synthetic) datasets are normalized to the 2D space $[0, 10000] \times [0, 10000]$.

ID	Contents	Data cardinality
PO	Political/Ocean	37945
PP	Populated Places	14581
RD	Roads	327561
RR	Railroads	44746
UT	Utilities	18708

Table 1. Real datasets of China

For a feature dataset $\mathcal{F}_c$, we generated qualities for its points such that they simulate a real world scenario: facilities close to (far from) a town center often have high (low) quality. For this, a single *anchor* point $s_\star$ is selected such that its neighborhood region contains high number of points. Let $dist_{min}$ ($dist_{max}$) be the minimum (maximum) distance of a point in $\mathcal{F}_c$ from the anchor $s_\star$. Then, the quality of a feature point s is generated as:

$$\omega(s) = \left(\frac{dist_{max} - dist(s, s_\star)}{dist_{max} - dist_{min}}\right)^\lambda \tag{6}$$

where $dist(s, s_\star)$ stands for the distance between s and $s_\star$, and λ controls the skewness of quality distribution. In this way, the qualities of points in $\mathcal{F}_c$ lie in $[0, 1]$ and the points closer to the anchor have higher qualities. Also, the quality distribution is highly skewed for large values of λ.

We study the performance of our algorithms with respect to various parameters, which are displayed in Table 2 (their default values are shown in bold). In each experiment, only one parameter varies while the others are fixed to their default values. Note that the parameter ϵ is applicable to range score queries only and query ranges for different feature datasets share the same ϵ.

Parameter	Values
Buffer size (%)	0.1, 0.2, **0.5**, 1, 2, 5, 10
Object data size, $\lvert\mathcal{D}\rvert$ ($\times 1000$)	100, **200**, 400, 800, 1600
Feature data size, $\lvert\mathcal{F}\rvert$ ($\times 1000$)	50, **100**, 200, 400, 800
Quality skewness, λ	0.5, **1.0**, 1.5, 2.0, 2.5
Number of results, k	**1**, 2, 4, 8, 16, 32, 64
Number of features, m	1, **2**, 3, 4, 5
Query range, ϵ	10, 20, **50**, 100, 200

Table 2. Range of parameter values

4.2 Performance Study

Objects	Features	I/O			
		SP	GP	BB	FJ
UT	RD, RR	9254	3329	303	306
RR	RD, UT	38804	4961	1100	1074
RD	RR, UT	1022582	15162	5434	216
RR	RD, PO, UT	132267	6139	3409	30097
RR	RD, PO, UT, PP	190939	7087	3692	90313

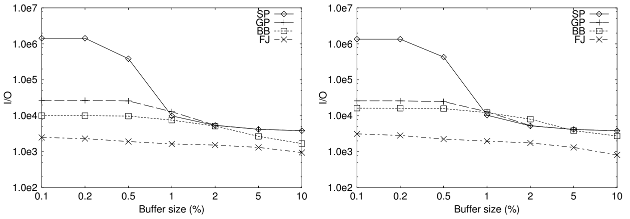
Table 3. Range score queries on China datasets

Objects	Features	I/O			
		SP	GP	BB	FJ
UT	RD, RR	10031	3363	527	52
RR	RD, UT	38852	4755	2152	1615
RD	RR, UT	1693053	21462	12579	227
RR	RD, PO, UT	243004	7147	4108	13194
RR	RD, PO, UT, PP	328862	8323	5391	69615

Table 4. NN score queries on China datasets

In the first experiment, we show the effect of data distribution on the cost of the algorithms, by choosing different combinations of the object dataset and feature datasets. Table 3 shows the cost of the algorithms for range score queries. Observe that SP, GP and BB follow the same trend, while FJ shows a different trend, because the I/O cost of methods SP, GP and BB heavily depends on the size of $\mathcal{D}$, while that of method FJ mainly depends on the sizes of the feature sets and the distribution of the qualities. For the cases of only two feature trees, FJ outperforms all the other methods, while for the cases where more than two feature trees are considered, FJ is more expensive than BB and GP. Table 4 shows the case for NN score queries, where the costs of the algorithms follow a similar trend as for range score queries.

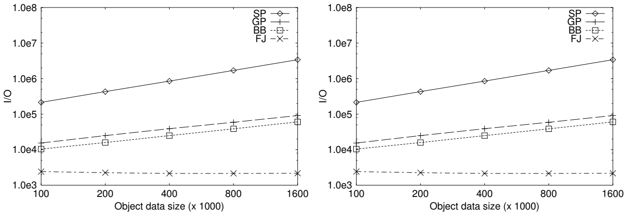
In subsequent experiments, we compare the cost of the algorithms on synthetic datasets with respect to different parameters. Figure 7 plots the cost of the algorithms as a function of the buffer size. As the buffer size increases, the cost of all algorithms drops and their performance gap shrinks. At low buffer sizes, GP is cheaper than SP because GP computes the scores of points within the same leaf node concurrently; however, it is more expensive than BB, since BB reduces score computation of points in leaf nodes by pruning non-leaf entries with upper bound scores smaller than γ. FJ outperforms its competitors as it discovers qualified combination of feature entries early.



(a) Range score (b) NN score

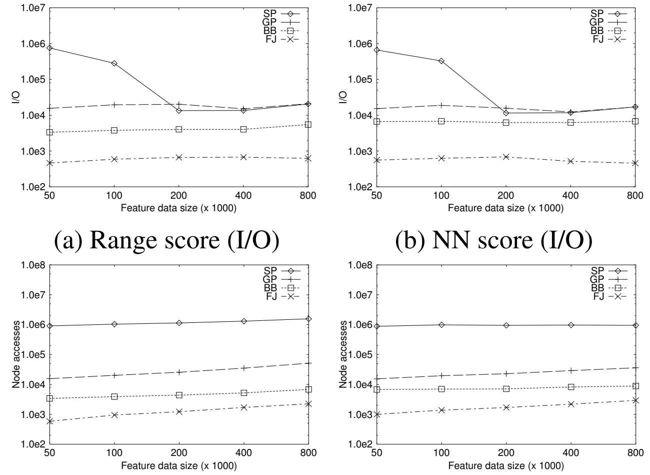
Figure 7. Effect of buffer size

Figure 8 compares the cost of the algorithms with respect to the object data size $|\mathcal{D}|$. Since the cost of FJ is dominated by the cost of joining feature datasets, it is insensitive to $|\mathcal{D}|$. On the other hand, the cost of the other methods (SP, GP, BB) increases with $|\mathcal{D}|$, as score computations need to be performed for more objects in $\mathcal{D}$.



(a) Range score (b) NN score

Figure 8. Effect of $|\mathcal{D}|$



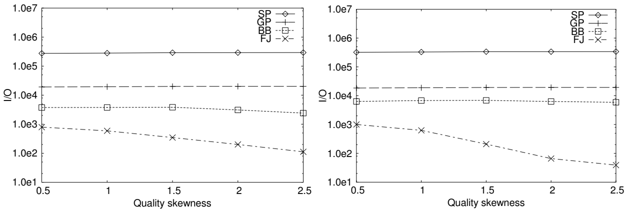
(a) Range score (I/O) (b) NN score (I/O)

(c) Range score (node accesses)(d) NN score (node accesses)

Figure 9. Effect of $|\mathcal{F}|$

Figure 9a,b (9c,d) plot the I/O cost (node accesses) of the algorithms with respect to the feature data size $|\mathcal{F}|$ (of each feature dataset). As $|\mathcal{F}|$ increases, the number of node accesses of all algorithms increases. On the other hand, the memory buffer size also increases (since it is fixed to 0.5% of the sum of tree sizes), so the algorithms have smaller I/O increase or even cost reduction.

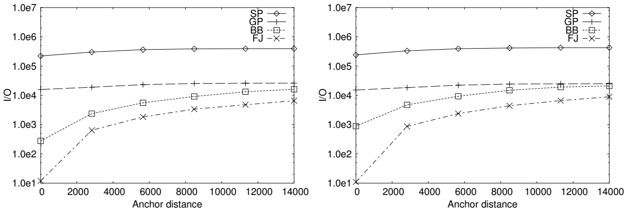
Figure 10 shows the cost of the algorithms as a function of the quality skewness λ. As we see, the costs of SP, GP and BB do not decrease a lot compared to that of FJ; for higher values of λ, FJ manages to locate the combinations that contribute to the top-k results early and easily prune high-level combinations that cover other regions of very low scores.



(a) Range score (b) NN score

Figure 10. Effect of λ

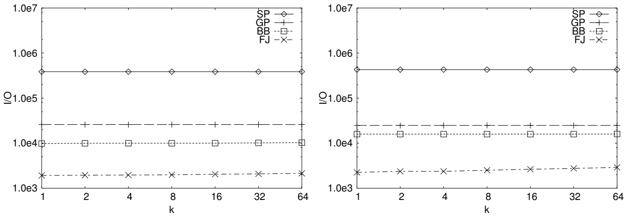
Figure 11 compares the cost of the algorithms by varying the distance between anchor points in feature datasets $\mathcal{F}_1$ and $\mathcal{F}_2$. The costs for all the methods increases with the distance, because the scores of the top-k results decrease and these results are spread to wider spatial regions; thus, more accesses are required to retrieve the results.



(a) Range score (b) NN score

Figure 11. Effect of anchor distance

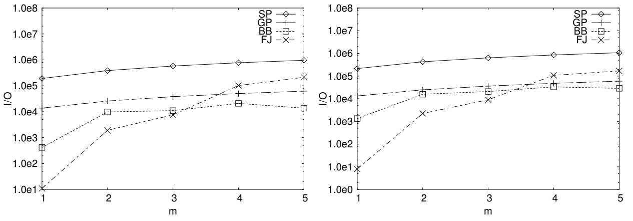
Figure 12 shows the cost of the algorithms as a function of the number k of requested results. Since SP and GP compute the scores for all objects in $\mathcal{D}$, their performance is independent of k. As k increases, both BB and FJ have weaker pruning power and their cost increases slightly.



(a) Range score (b) NN score

Figure 12. Effect of k

Figure 13 plots the cost of the algorithms with respect to the number m of feature datasets. The costs of SP, GP and BB increase slowly as m increases because they apply the incremental computation technique (see Section 3.1) to reduce the number of component score computations. Thus, disqualified points can be pruned without computing all their component scores. On the other hand, the cost of FJ increases significantly with m, because the number of qualified combinations of entries is exponential to m.



(a) Range score (b) NN score

Figure 13. Effect of m

Finally, Figure 14 shows the cost of the algorithms for range score queries, when varying the query range ϵ. As ϵ increases, SP accesses more nodes in feature trees to compute the scores of the points. Although the same happens for the other methods, they have lower I/O cost, as the buffer absorbs better the effect of ϵ. Summing up, FJ is the best method for two or fewer feature sets, while BB should be used for more than two features in the query.

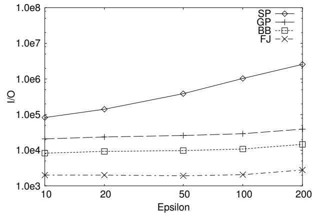


Figure 14. Effect of ϵ, range score

5 Discussion

The top-k spatial preference queries we studied in this paper can be generalized to include additional components that could be useful in practice. First, in the definition of $\tau_c^\theta(p)$, we can use a *weighted* version of $\omega(s)$, based on $dist(p,s)$. In range score computation, the quality $\omega(s)$ of a feature s is multiplied by the weight $\psi = 1 - \frac{dist(p,s)}{\epsilon_c}$, and the feature with the maximum weighted quality is considered in the component distance. Note that the weight ψ is always a value in the interval $[0,1]$. In NN score computation, we weigh based on a maximum acceptable distance of a nearest neighbor. Thus, data objects with very close nearest neighbor of moderate quality may be preferable to objects with a far nearest neighbor, even if it has a higher quality. Our algorithms can directly be applied for such extended preference queries. A simple optimization applicable for these queries is to prune non-leaf entries at feature trees, if they are closer than ϵ_c, but cannot improve γ, due to the distance of their MBRs to the examined objects.

Another extension of our queries include preferences also on the data objects. For instance, assume that we are looking for flats that are cheap, big, and close to vegetarian restaurants. There are several ways to process such queries. First, our algorithms can be used for this purpose if the object tree is also an aR-tree, where the non-spatial attributes (e.g., price, size) are aggregated accordingly. BB can be optimized to prioritize the examination of data in $\mathcal{D}$, based on preferences on the object attributes and prune subtrees that cannot lead to better results than the ones currently found. Another method is to search primarily on the non-spatial preferences, with the use of a top-k algorithm [12] and probe the feature sets for the spatial preference component, as long as the current top-k results can be improved. In the future, we plan to study in detail the optimization of such queries.

6 Conclusion

In this paper, we studied top-k spatial preference queries, which provides a novel type of ranking for spatial objects based on qualities of features in their neighborhood. We presented several algorithms for processing top-k spatial preference queries. First, we introduced a baseline algorithm SP that computes the scores of every object by performing spatial queries on feature datasets. SP is optimized by an incremental computation technique that reduces the number of component score computations for the objects. Next, we presented the GP, a variant of SP that reduces I/O cost by computing scores of objects in the same leaf node concurrently. Based on GP, we developed algorithm BB, which prunes non-leaf entries in the object tree that cannot lead to better results. For this, we developed techniques for deriving upper bound scores for non-leaf entries in the object tree by accessing feature trees. Finally, we propose algorithm FJ, which performs a multi-way join on feature trees to obtain combinations of feature points that commonly affect a spatial region and then search for the objects (in the object tree) affected by these combinations.

Our experimental results show that BB outperforms SP and GP, since SP and GP examine every object in the object tree, whereas BB applies pruning techniques to reduce the number of objects to be examined (and thus their score computations). FJ is more efficient than BB for two or less feature sets, because FJ effectively discovers combinations of features that may lead to results with high scores. BB and FJ mainly access object data and feature data, respectively. Thus, BB is the best method when the object dataset is small whereas FJ is the best algorithm when there are few and small feature datasets.

Apart from the problem variants discussed in Section 5, in the future, we plan to design a cost model for BB and FJ so that the query optimizer is able to decide the best query algorithm (either BB or FJ) for a particular problem input. Another interesting research direction is to investigate efficient processing of top-k spatial preference queries on non-indexed data.

References

[1] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[2] S. Berchtold, C. Boehm, D. Keim, and H. Kriegel. A Cost Model for Nearest Neighbor Search in High-Dimensional Data Space. In *PODS*, 1997.

[3] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *ICDT*, 1999.

[4] S. Brin and L. Page. The Anatomy of a Large-scale Hypertexual Web Search Engine. In *WWW*, 1998.

[5] N. Bruno, L. Gravano, and A. Marian. Evaluating Top-k Queries over Web-accessible Databases. In *ICDE*, 2002.

[6] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient Query Processing in Geographic Web Search Engines. In *SIGMOD*, 2006.

[7] Y. Du, D. Zhang, and T. Xia. The Optimal-Location Query. In *SSTD*, 2005.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.

[9] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

[10] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *TODS*, 24(2):265–318, 1999.

[11] I. F. Ilyas, W. G. Aref, and A. Elmagarmid. Supporting Top-k Join Queries in Relational Databases. In *VLDB*, 2003.

[12] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung. Efficient Aggregation of Ranked Inputs. In *ICDE*, 2006.

[13] N. Mamoulis and D. Papadias. Multiway Spatial Joins. *TODS*, 26(4):424–475, 2001.

[14] A. Okabe, B. Boots, K. Sugihara, and S. Chiu. *Spatial Tessellations : Concepts and Applications of Voronoi Diagrams*. Wiley, second edition, 2000.

[15] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP Operations in Spatial Data Warehouses. In *SSTD*, 2001.

[16] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *SIGMOD*, 1995.

[17] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-Bound Processing of Ranked Queries. *Information Systems*, to appear.

[18] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[19] T. Xia, D. Zhang, E. Kanoulas, and Y. Du. On Computing Top-t Most Influential Spatial Sites. In *VLDB*, 2005.

[20] D. Zhang, Y. Du, T. Xia, and Y. Tao. Progessive Computation of The Min-Dist Optimal-Location Query. In *VLDB*, 2006.