Efficient Skyline Computation in Metric Space

David Fuhry Ruoming Jin
Department of Computer Science
Kent State University
{dfuhry,jin}@cs.kent.edu

Donghui Zhang
College of Computer & Information Science
North East University
donghui@ccs.neu.edu

Abstract

Given a set of n query points in a general metric space, a metric-space skyline (MSS) query asks what are the closest points to all these query points in the database. Here, consider for any point p, if there are no other points in the database which have less or equal distance to all the query points, then p is denoted as one of the closest points to the query points. This problem is a direct generalization of the recently proposed spatial-skyline query problem, where all the points are located in two or three dimensional Euclidean space. It is also closely related with the nearest neighbor (NN) query, the range query and the common skyline query problem. In this paper, we have developed new algorithms to aggressively prune non-skyline points from the search space. We also contribute two new optimization techniques to reduce the number of distance computations and dominance tests. Our experimental evaluation has shown the effectiveness and efficiency of our approach.

1. Introduction

In many applications, similarity search is more practical than exact match. For instance, in **image retrieval**, there may not exist any image in the database which is exactly the same as the query image. To increase the search accuracy, one may use multiple query images, such as those extracted from a video taken during a crime. Different query images may be the most similar to different images in the database. The question is: what are the most similar images to ALL the query images? The skyline concept can be used here to answer the above question. In particular, the query result should be the images in the database which are not dominated by any other image. Image A is dominated by image B, if B is more (or equally) similar to all query images than A is. Here the images are mapped to a metric space and the similarity between two images is often captured by the Hausdorff metric [14]. The similarity search problem with multiple images is an instance of the *metric-space skyline (MSS) query*.

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

There is a fundamental difference between the traditional skyline query and the MSS query. In the traditional skyline query, each object is a row in a relational database table which can be mapped to a multi-dimensional point. Spatial access methods like the R-tree can be used to index these points and to enable efficient skylining algorithms like Branch and Bound Skyline (BBS) [21]. The spatial skyline query (SSQ) [27], which also involves multiple query points, relies on the existence of a multi-dimensional space to derive Voronoi-diagram or convex-hull based algorithms. In the MSS query, the objects are mapped into a metric space, not a multi-dimensional space. Without a multi-dimensional space, the MSS query cannot be solved efficiently using existing multidimensional skylining algorithms. In general metric space, there is no minimum bounding rectangle (MBR) as used in BBS or space partitioning as used in SSQ.

There are many other real applications of the MSS query besides image retrieval, such as **DNA searching**. DNA microarray experiments allow biologists to quickly identify many groups of co-expressed or co-repressed genes at different experimental conditions. Suppose the post-analysis identifies a set of tens or even hundreds of candidate genes related to a certain disease. A key operation is then to identify the genes which share the strongest sequence similarity to the set of candidate genes from the DNA sequence database. As the DNAs can be mapped to a metric space where the sequence similarity is commonly described by a certain *edit* distance [20], the DNA searching problem is another instance of the MSS query.

1.1 Problem Definition

DEFINITION 1 (METRIC SPACE). *A* **metric space** *is a pair (D, d), where D is a set of points and d is the distance function: $d : D \times D \rightarrow R$ (R is the real domain) which satisfies the following properties for any p, q, $s \in D$:*

1. **Positivity & Identity***: $d(p, q) \geq 0$ and $d(p, p) = 0$*
2. **Symmetry***: $d(p, q) = d(q, p)$*
3. **Triangle Inequality***: $d(p, q) + d(q, s) \geq d(p, s)$*

A metric space is also called a metric-space database since D stores a set of objects. Consider a metric-space database (D, d). Two basic similarity queries are:

- **NN(q):** Given a query point q, return the closest point in D to q.

- **Range(q,r):** Given a query point q and a distance $r > 0$, return all the points in D whose distances to q are no more than r.

DEFINITION 2 (DOMINANCE). *Consider a metric-space database (D, d), and a set of query points $Q = \{q_1, q_2, \ldots, q_m\}$. For any two points p, $p' \in D$, p* **dominates** *p' if the following two conditions hold:*

1. $\forall q_i \in Q$, $d(p, q_i) \leq d(p', q_i)$;
2. $\exists q_j \in Q$, $d(p, q_j) < d(p', q_j)$.

Intuitively, p dominates p' if p has an equal or smaller distance than p' to all query points in Q, and p has a smaller distance than p' to at least one query point.

DEFINITION 3 (METRIC-SPACE SKYLINE). *Consider a metric-space database (D, d), and a set of query points $Q = \{q_1, q_2, \ldots, q_m\}$. The* **metric-space skyline (MSS)** *is the set of objects in D which are not dominated by any other object in D.*

This paper addresses the problem of computing metric-space skylines.

1.2 Challenges of the MSS Query

For a large database, indexed skyline computation is a must, as non-indexed algorithms, which need $O(|D|^2)$ dominance tests and $|D||Q|$ distance computations, are very inefficient. However, as discussed before, existing indexed skyline computation like BBS or SSQ assumes a multi-dimensional vector space and therefore does not apply to the MSS query.

So the first challenge for us is how to design efficient algorithms based on metric-space indices like the *Vantage-Point tree (VPT)* [30, 32]. In addition, even though the general idea of BBS can apply to MSS, it will be inefficient without utilizing the properties of the general metric space, such as the triangle inequality.

The second challenge is how to minimize the number of distance computations and dominance tests. In a metric space, distance computation can be very expensive [5]. Similarly, a dominance test can also be very expensive. For instance, a dominance test involving two data points with respect to a set of m query points can require m comparisons (one comparison per query point). As the number of query points can be relatively large (tens or even hundreds), the cost of dominance tests can become the bottleneck of the search process.

1.3 Recent Work

Very recently and concurrent with our work [13], Chen and Lian [8] introduced a straightforward adaptation of BBS for metric space datasets indexed by M-Tree, introducing the MSQ algorithm to compute the spatial skyline. Unlike their work, we make a systematic effort to reduce the number of distance computations and dominance tests, using two new optimization techniques. These *dynamic indexing* and *k-dispersion* extensions are detailed in Section 3, and it is shown that they can significantly reduce dominance tests and distance computations.

1.4 Our Contributions

In this paper, we design algorithms to aggressively utilize the basic properties of the metric space, i.e., the triangle inequality, and incorporate the existing indexes for metric space, such as GHT (*Generalized-Hyperplane* tree) [30], VPT (*Vantage-Point* tree) [30, 32], SAT (*Spatial-Approximation* tree) [19], M-tree [11] and their variants including MVPT (*Multiple-Vantage-Point* tree) [3], to speedup the processing of the MSS query. We have revealed some interesting relationships between the existing similarity queries, i.e. NN (Nearest Neighbor) and range queries, and MSS queries. Simply speaking, we show that an appropriate range query of any query point q with its range being a function of its distance to its nearest neighbor provably contains all the resulting points for the MSS query. This property can help us immediately reduce the search space for MSS queries significantly. Further, we have developed techniques to effectively prune different metric-space constructs, such as regions formed by covering balls and general hyperplanes, which do not contain any MSS points.

We propose two new optimization techniques: *dynamic indexing* and *k-dispersion*, to reduce dominance tests and distance computations, respectively. *Dynamic indexing* organizes the already-computed MSS points into buckets such that the candidate MSS points can be quickly pruned if they are dominated by some other points. This approach is similar to the index approach in [29] for the general skyline query; however, our index is dynamically maintained for the answering set. The *k-dispersion* optimization uses a subset of k maximally-distant [12] query points to facilitate distance estimation. For a candidate MSS point p, only its distances to the chosen k query points are calculated. The other distance values are estimated using triangle inequality and the known pairwise distances between query points. A *lazy* technique computes exact values only when they are needed later. We note that the k-dispersion points serve as the analog to vertices of the convex hull of the query point in Euclidean space.

We have performed a detailed experimental evaluation on both real and synthetic datasets. Our experimental results show our algorithms and techniques can significantly speed up the processing of MSS queries and reduce the number of distance computations and dominance tests. Overall, the performance gain can be more than 300 times compared with the naive method.

2. Metric Skyline Algorithms

In this section, we present two algorithms for efficiently processing MSS queries. Subsection 2.1 presents N^2RS (Nearest-Neighbor-Range-Skyline), which uses familiar nearest neighbor (NN) and range queries, and requires two passes of the database. Subsection 2.2 introduces B^2MS^2 (Branch-and-Bound Metric Space Skyline), which can progressively return MSS points in a single pass of the database. Subsection 2.3 generalizes the typical metric-space enclosing ball geometry into the $GMBR$ (General Minimal Bounding Region), which allows a broader range of metric space indexes to be utilized in the B^2MS^2 algorithm.

2.1 Answering MSS queries using NN and Range Query

We start with two lemmas describing the basic relationships between NN, Range query, and MSS query.

LEMMA 1. *Given a metric-space database D and a set of query points Q, for any $q \in Q$: 1) if q has only one nearest neighbor $p \in D$, then then p is a MSS point; and 2) if q has more than one nearest neighbor, then at least one of them is a MSS point.*

Proof: Case 1: If p is the only nearest neighbor of q, then, for any other point p', $d(p, q) < d(p', q)$. Thus, no point can dominate q, and p is a MSS point.

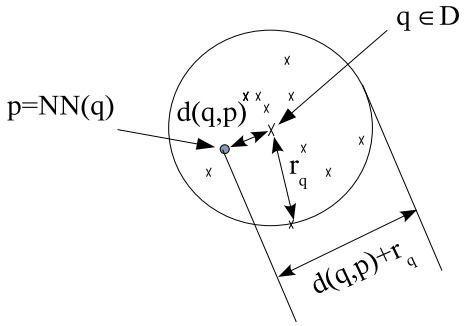


Figure 1. NN, Range, and MSS queries

Case 2: If there are multiple nearest neighbors of q, then no other point can dominate them (based on the argument in Case 1). Therefore, there is at least one of nearest neighbors is a skyline point based on the transitivity of the skyline points. □

This lemma provides a stronger result than the results in [27], which considers only the cases in the Euclidean space and does not separate these two cases. This lemma suggests for any $q \in Q$: $NN(q) \cap MSS(Q) \neq \emptyset$

Note that in the general case, it is not true that $NN(q) \subseteq MSS(Q)$.

The following lemma builds the relationship between NN, the range query and the MSS query. To facilitate our discussion, we define the *enclosing ball* as a pair (p, r_p) for a set of points X, $r_p = max_{x \in X} d(x, p)$. We refer p as the center of the ball, and r_p is the radius of the ball. Note that in the general case, p is not necessarily a point in X. We will specify if $p \in X$.

LEMMA 2. *For any point q in the query set Q, let (q, r_q) be an enclosing ball for Q. The distance between a MSS point and q is bounded by $2r_q + d(q, NN(q))$.*

Proof: For a point $q \in Q$, the furthest distance from q's nearest neighbor to any point in Q is bounded by $d(q, NN(q)) + r_q$ (Figure 1). Suppose a point p' is outside the radius of $r_q + d(q, NN(q)) + r_q$: $d(p', q) > 2r_q + d(q, NN(q))$, then for any query point q',

$$d(p', q') > r_q + d(q, NN(q)) \geq d(NN(q), q')$$

Therefore, the point p' is dominated by $NN(q)$ and is not a MSS point. Therefore, The distance between a MSS point and q cannot be more than $2r_q + d(q, NN(q))$. □

This lemma suggests that for any $q \in Q$: $MSS(Q) \subseteq Range(q, 2r_q + d(q, NN(q)))$

Utilizing this lemma, we can process a MSS query as follows. We select a query point q (the selection will be discussed later), and find the nearest neighbor of q (NN(q)). Then, we perform a range query $Range(q, 2r_q + d(q, NN(q)))$. For each answering point in the range query, we compute their distance to all the points in Q, and generate a $|Q|$-dimensional database. Finally, we apply a general skyline algorithm, such as BBS, to identify the skyline points in the new space. Based on the definition, these points are our targeted MSS points. Algorithm 2.1 is the sketch of this algorithm.

Note that in the N^2RS algorithm, we use a heuristic (smallest-enclosing-ball) to choose the center. It will help us to minimize the first term ($2r_q$) of the range in the range query (Line 3, Algorithm 2.1). The second term is hard to minimize before we find all the nearest neighbors for each query point. Note that this heuristic basically targets minimizing the total number of candidate MSS points.

Algorithm 1 $N^2RS(D, Q)$

Parameter: A metric space dataset D
Parameter: A nonempty set of query points Q
Condition: The enclosing ball $(q, r_q), q \in Q$ for Q chooses the query point with the smallest radius
1: $(q, r_q) \leftarrow$ Smallest-Enclosing-Ball for Q
2: $p \leftarrow NN(q)$
3: $R \leftarrow Range(q, 2 \times r_q + d(p, q))$
4: $S \leftarrow \emptyset$
5: **foreach** $p \in R$ **do**
6: $\quad S \leftarrow S \cup \{(d(p, q_1), \cdots, d(p, q_m)\}$
7: **end for**
8: **return** Skyline(S);

The computational complexity for N^2RS is as follows. In the worst case, the computational cost for NN and Range query are $O(|D|)$. The cost of traditional skyline algorithm is bounded by $O(|R|log|R| + |R||Q|)$, where the size of the range query result is $|R|$. The first term is the cost for sorting or building an R-tree. The second term is the cost for dominance test. Put together, the total cost of above algorithm is $O(|D| + |R|(log|R| + |Q|))$. The number of distance computations is $O(|Q|^2 + |D| + |R||Q|)$, where $|Q|^2$ is the computational cost of finding the smallest enclosing ball, $|D|$ is for the nearest neighbor and range query, and $|R||Q|$ is for construction of the new space.

Even though the N^2RS algorithm significantly reduces the computational cost for the MSS query compared with the naive solutions, there are several drawbacks of this approach. First, we have to search the entire database twice, one for the nearest neighbor query and another for the range query. Second, the entire range query results has to be mapped for a new space for utilizing the general skyline algorithms. Thus, such algorithms will not be able to utilize the metric space properties of these points, and therefore, can be inefficient. Finally, the user has to wait until the completion of the range query to get any searching results.

2.2 B^2MS^2:A Progressive Algorithm for MSS queries

In the following, we introduce a progressive (or online) algorithm called B^2MS^2 (Branch-and-Bound Metric Space Skyline), which can quickly return the first MSS points without having to read the entire database. It will scan only the database once and without explicitly transforming to a new multi-dimensional space. Aggressive pruning of the non-MSS points is exploited based on triangle inequality. In addition, many different existing metric-space indexes can be utilized in the algorithm.

Generic Index Tree: The generic power of this algorithm is its simple assumption for an index structure. It has two basic requirements:

1. The index structure is a tree where each node records an enclosing ball to cover all its point.

2. Each leaf node contains a single point.

For most of the index structures, both requirements can be met easily. For the first one, we need an *enclosing ball* to cover all

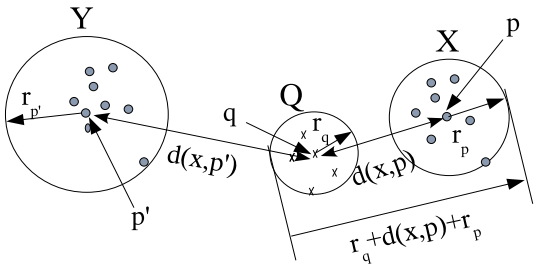


Figure 2. Lower Bound and Upper Bound

the points of each node in the index. Such information is already recorded into most of the indexes, for instance, the GHT, SAT, M-tree etc. In the next subsection, we will introduce a generalization of the enclosing ball, which can handle other variations (VP, MVP, etc). For the second requirement, if each leaf node contains a set of points, we can just (conceptually) add another level of the nodes each recording a single point. We note that Chen and Lian [8] recently developed an algorithm MSQ with similar ideas for only the M-Tree metric space index. Our algorithm targets any of the other metric index types listed above.

Before we describe the algorithms, we will take a detailed look at the generic index used in B^2MS^2: GHT. This index is based on the definition of a generalized hyperplane in metric space.

DEFINITION 4. *[30] Given two points p_1 and p_2, a generalized hyperplane (GH) is the set of points in the database D that are equally close to these two points:*

$$GH(p_1, p_2) = \{p | d(p, p_1) = d(p, p_2), p \in D\}$$

Suppose we randomly choose two points p_1 and p_2 in the database. The entire database D can then be partitioned by their hyperplane. One partition will be the set of points which are closest to p_1; the others are closest to p_2 (points exactly on the hyperplane can belong to either set).

A GHT essentially utilizes the hyperplane to recursively partition the database and build a binary tree accordingly. For searching and querying purposes, each node in the binary tree records an enclosing ball, where the points for the general hyperplane serve as the center of the ball. Leaves of a GHT record a single point. Note that different heuristics/methods can be used for building a balanced tree and for extending it to a k-way partition, for example in GNAT [5]. Here, N^2RS assumes that such an index is already constructed.

Lower Bound and Upper Bound: Here, we introduce two bounds associated with each node in the index tree. Such bounds form the basis for B^2MS^2. First, given the enclosing balls (q, r_q) and (p, r_p) for the query set Q and a node X in the index tree, respectively, we introduce the lower bound and upper bound for the node X as follows:

Lower Bound: The lower bound is the closest possible distance between a point in X and a point in Q and is estimated using the enclosing balls. We denote the lower bound as $LB(X, Q)$: $LB(X, Q) = d(q, p) - r_q - r_p$

Upper Bound: The upper bound is the furthest possible distance between a point in X and a point in Q and is estimated using the enclosing balls. We denote the upper bound as $UB(X, Q)$: $UB(X, Q) = d(q, p) + r_q + r_p$

The following lemma describes a simple way to prune a node if it does not contain any MSS point.

LEMMA 3. *Suppose we have two nodes X and Y, if $UB(X, Q) < LB(Y, Q)$, then Y will not contain any MSS point.*

Proof: Figure 2 shows the scenario. Consider a point x in X. Its distance to any query point q' will be smaller than the distance between any point y in Y and q': $d(x, q') \leq UB(X, Q) < LB(Y, Q) \leq d(y, q')$. Thus, all the points in Y will be dominated by x. Therefore, Y will not contain any MSS point. □

This lemma can help quickly shrink the search range for the MSS points and aggressively prune unpromising nodes. In particular, we note that if we have a collection of nodes, $X_1, \cdots, X_m$, then we can apply the minimum of the their upper bounds for pruning. In other words, we can define the upper bound for a collection of nodes: $LB(\{X_1, \cdots, X_m\}, Q) = min_{1 \leq i \leq m} LB(X_i, Q)$

Algorithm 2 $B^2MS^2(T, Q)$

Parameter: The root node of the index tree T
Parameter: A nonempty set of query points Q

```
1: (q, r_q) ← Smallest-Enclosing-Ball for Q
2: S ← ∅ // Resulting Set
3: T.mindist ← LB(T, Q)
4: H.push(T) // priority queue ordered by mindist
5: outer_bound ← ∞ // the Upper Bound
6: while H ≠ ∅ do
7:    p ← H.pop()
8:    if p.isLeafNode() then
9:       if !skyline_dominates(S, p, Q) then
10:         S.push(p)
11:      end if
12:   else
13:      foreach p' ∈ p.children do
14:         if LB(p', Q) ≤ outer_bound then
15:            p'.mindist ← LB(p', Q)
16:            H.push(p')
17:            outer_bound ← min(outer_bound,
18:                UB(p', Q)
19:         end if
20:      end for
21:   end if
22: end while
23: return S
```

B^2MS^2 **Algorithm Description:** The B^2MS^2 algorithm assumes a generic index tree is constructed for the database D. Algorithm 2 is the sketch of B^2MS^2. For a given query set Q, it maintains a heap which records all the nodes that potentially include MSS points. The nodes are ordered by their lower bounds ($p.mindist = LB(p, Q)$, Line 3 and 15). In addition, we maintain the minimum of all upper bounds for pruning (Line 17). We also maintain a set S recording partially computed MSS points (Line 2). We always retrieve the first node in the heap (Line 7). If the node is a leaf (single point), we perform the dominance test for it: compare it with all the confirmed MSS points (Line 8). We note that if there is a tie of lower bounds, we will perform a pair-wise dominance test for all the points sharing the tie. We drop points which are dominated. For undominated points, we compare them with the MSS point in S. If the node is an intermediate node, we will see it can be pruned (Line 14). If it cannot be pruned, we will retrieve all its children, and insert them into the heap by their lower bound (Line 15). The algorithm proceeds until the heap is

empty.

The following invariants of the algorithm show the interesting relationship between B^2MS^2 and N^2RS.

LEMMA 4. *Let (q, r_q) be the enclosing ball for the query set Q. The first MSS point being inserted into S is q's nearest neighbor* [1]. *Further, the $outer\_bound$ is bounded by $d(q, NN(q))+r_q$.*

Thus, we can see that the algorithm will quickly find q's nearest neighbor and then continue the range query process. However, unlike N^2RS, B^2RS^2 will access each node in the index tree only once. The utilization of an outer-bound allows us to efficiently prune nodes that cannot contain skyline points. Based on the lemma, we can see that the number of candidate points being tested for dominance is $|R|$, where R is the resulting set of the range query $R(q, 2r_q + d(q, NN(q)))$. Thus, the worst case computational complexity of the algorithm is $O(|D| + |R||Q|)$. The number of distance computations is $O(|Q|^2 + |D| + |R||Q|)$, where $|Q|^2$ is the computational cost of finding the smallest enclosing ball, $|D|$ is for scanning the entire index tree, and $|R||Q|$ is for performing the dominance tests.

The correctness of the algorithm is as follows. For any point p' being extracted from the heap, all the points that are extracted earlier had a smaller lower bound. This is equivalent to saying that each of those points had a smaller distance to q. Thus, each of them will not be dominated by p'. For the same reason, any point being extracted later than p' has a larger distance to q. Therefore, a later point can not dominate p'. Given this, this algorithm will correctly output all the MSS points.

Finally, we note that both algorithms can require a significant number of distance computations and dominance tests. This will be addressed in Section 3.

2.3 Utilizing other Metric Space Indexes

In this subsection, we generalize the enclosing ball notation for covering all the points of nodes in an index tree. Such a structure can be more flexible for utilizing existing indexes, such as VP-tree [32] and MVP-tree [3], and improve the pruning performance. The new notation is referred to $GMBR$ (General Minimum Bounding Region), and is represented as

$$\{p|l_1 \leq d(p, x_1) \leq h_1 \wedge \cdots \wedge l_n \leq d(p, x_n) \leq h_n\}$$

Here, the $x_1, \cdots, x_n$ do not have to be in the set of database points. Note that the ball is a special case of above representation. Consider the ball (q, r_q) for a set of points. It is equivalent to a 1-dimensional GMBR, where $x_1 = q$ and $l_1 = 0, h_1 = r_q$. A VP-tree node is another special-case of the GMBR, where $x_1 = x_2 = \cdots = x_n$ and $h_1 = l_2, h_2 = l_3, \cdots, h_{n-1} = l_n$.

Now we consider how to prune such a region. We will utilize the lower bound and upper bound of the distance between any two sets of points. The following theorem identifies these two bounds. Although further details are omitted due to space constraints, we note that this theorem provides the basis for pruning nodes in VP- and MVP-trees.

THEOREM 1. *For a given node X, the lower bound between a point in the query set $Q(q, r(q))$ and the point in $GMBR$ is $LB(X, Q) = max_{1\leq i\leq n} max(d(q, x_i) - h_i - r_q, l - d(q, x_i) - r_q)$. The upper bound is $UB(X, Q) = min_{1\leq i\leq n} d(q, x_i) + r_q + h_i$.*

[1] If it has more than one, it will be one of them.

3. Optimization Techniques for Distance Computation and Comparison

This section focuses on studying two unique challenges arising in MSS queries which have not been adequately studied before: the cost of distance computations and dominance tests. The number of distance computations for N^2RS is bounded by $O(|Q|^2 + 2N + |Q||R|)$ and for B^2MS^2 is bounded by $O(|Q|^2 + N + |Q||R|)$ (where $|R|$ is the number of leaf nodes not pruned). Both will require $O(|R|^2)$ dominance tests and therefore $O(|R|^2|Q|)$ distance comparisons. Our goals here are to reduce both the number of distance computations and the number of dominance tests.

3.1 Reducing the number of dominance tests

Given a candidate skyline point and a collection of skyline points S, to reduce the number of dominance tests, we would like to organize S so that we can quickly prune the candidate skyline point if it is not a skyline point. A key observation is as follows: a non-skyline point is likely to be dominated by the skyline points which are close to its closest query point. In other words, a skyline point is not likely to dominate non-skyline points if they are closest to some query point other than the skyline point's closest query point. A procedure utilizing this observation for dominance tests is sketched in Algorithm 3.

We organize the points of S into $|Q|$ buckets, each bucket corresponding to a query point. Each of the skyline points is put into the bucket of its closest query point. The contents of each bucket are sorted according to their distance to their closest query point. Then for each point p, the query points are sorted according to their distance to p. We will first perform a dominance test against the points in the bucket of the closest query point, then those in the bucket of the second closest query point, and so on. We iterate through each bucket's points in their defined order. We will first compare p with the skyline points which are closest to the current query point q. Finally, we consider when we can skip the rest of the list in the bucket. Let $d(p, q)$ be the distance from p to its closet query point q. Suppose we are working on the bucket of q'. If for one of the bucket's skyline points p', $d(p', q') > d(p, q)$, then we can simply stop searching points ordered after p' in the bucket. The correctness of this procedure can be deduced by Lemma 5.

LEMMA 5. *For any point p'' in the bucket after p', if $d(p', q') > d(p, q)$, then p'' will not dominate $d(p, q)$.*

3.2 Reducing the number of distance computations

Each candidate skyline point will require $|Q|$ distance computations for a dominance test, where $|Q|$ is the number of query points. We consider how to reduce the number of such distance computations for each candidate point.

The basic idea to reduce the number of distance computations is utilizing the triangle inequality to estimate the distance. Here, for each candidate point, we will compute its distance to only several query points, and then use the computed distances to estimate its distances to the rest of the query points. The problem is twofold. The first one is determining which query points should be selected as the seeds for estimating other distance. The second problem is how to do the estimation.

Our proposed heuristic is to choose k representative points to

Algorithm 3 $skyline\_dominates(S, p, Q)$

Parameter: A set of skyline points S, iteratable by mindist
Parameter: A candidate skyline point p
Parameter: A set of query points Q
Returns: Whether any $s \in S$ dominates p with respect to Q

```
Order q ∈ Q such that d(p, q_1) ≤ d(p, q_2) ≤ ··· ≤ d(p, q_m)
for i = 1 to m do
    P ← q_i.bucket // All the skyline points in q_i in ascending order of their distance to q_i
    for j = 1 to |P| do
        if d(P[j], q_i) > d(p, q_1) then
            break // Skip the rest of the list
        end if
        if P[j] dominates p then
            return true
        end if
    end for
end for
return false
```

cover the set of all query points ($k > 1$) [2]. We intend for them to serve as the analog to the vertices in the convex hull of all the query points if they were in Euclidean space. For a given set of points, the k-dispersion points are those points where the sum of their mutual distance is maximal out of all the k points in the set [25]. We expect these k points to be an effective representation of Q with respect to pruning. When we apply triangle inequality to a candidate skyline point, a key factor is to have at least one query point which it is close to.

The k-dispersion problem is known to be NP-hard. Since the number of query points can be rather large, full enumeration will not work. Instead, we apply a simple greedy algorithm due to Ravi et. al [25]. Essentially, this algorithm first chooses two furthest points. Then it iteratively adds a new point to the selected set such that its distance to the already-selected points is maximal.

The second question is how to estimate the uncomputed distance. The direct solution is to apply each point to estimate a lower bound and then choose the maximal one. However, this procedure can be rather costly if k is big. A heuristic procedure can be used when k is big is to choose this point's closest query point (selected), and use that one to do the estimation.

We note that after we apply the lower bound, if we find it cannot be pruned by the current skyline point, we will immediately compute its true distance. This overhead will increase the overall cost of a dominance test. A related problem is that since we only compute the distance to k query points, we will not be able to rank a database point with respect to all query points (in the Algorithm 3). Our solution is as follows. We only build $k < m$ buckets instead of m, and all the skyline points are put into whichever of the k buckets they are closest to. The pruning process is similar to Algorithm 3.

4. Experimental Results

We seek to answer the following performance-related questions:

1. How do the three skyline algorithms, N^2RS, B^2MS^2, and MSQ perform on different datasets?
2. Which indexing structures perform best for these types of queries?
3. How effective are the optimizations proposed in Section 3?

[2] If $k = 1$, we will simply choose the medioid of the query point set.

Datasets: We use a total of 4 publicly available datasets which have been used in previous metric space studies. The 2 real datasets are a DNA sequence [3] used by the Mobios project [18] and a dictionary listing of single words in the English language from Project Gutenberg [31]. The 2 synthetic datasets are random 5-dimensional vectors and uniform 20-dimensional vectors, both from the Mobios project. Dataset characteristics are presented in Table 1.

4.1 Testing Parameters:

Each of our figures compares the performance of four metric skyline algorithms: N^2RS, B^2MS^2, MSQ, and a brute-force linear scan. The implementations of N^2RS and B^2MS^2 are our own, while the implementation of MSQ was generously provided to us by the authors of [8].

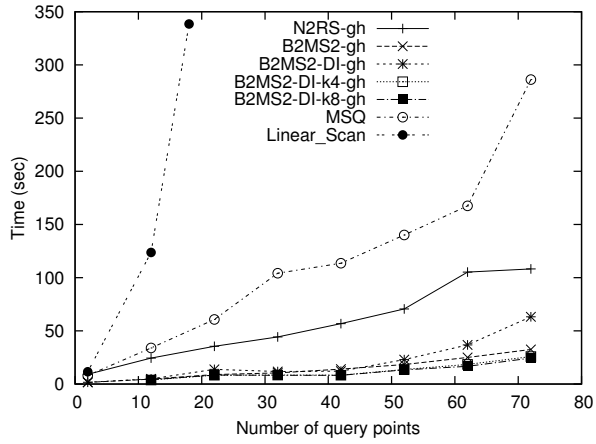
Among the metric indexes we implemented, GH-tree is most easily compared to M-Tree (used by MSQ), so we show GH-tree results for all four datasets. We also show VP-tree results for the dictionary and uniform-20d datasets, and SA-tree results for the random-5d dataset.

The branching factor, which only applies to the N^2RS and B^2MS^2 tests, is the maximal number of children of an intermediate node in a metric tree. For the dna, random-5d, uniform-20d, and dictionary datasets, we used branching factors of 40, 7, 20, and 45, respectively. The block size, which only applies to the MSQ test, is the size of M-Tree blocks into which sibling data items fit. We used block sizes of 1KB for the random-5d dataset and 10KB for the other datasets. The eight skyline parameter combinations we present are: 1) N^2RS using a GH-tree index; 2) B^2MS^2 using a GH-tree index; 3) B^2MS^2 using a VP-tree index; 4) B^2MS^2 using an SA-tree index; 5)B^2MS^2 using a GH-tree index and dynamic indexing; 6) B^2MS^2 using a GH-tree index, dynamic indexing, and a k-dispersion (of between 3 and 10); 7) MSQ using an M-Tree index; and 8) A brute-force, linear scan skyline computation.

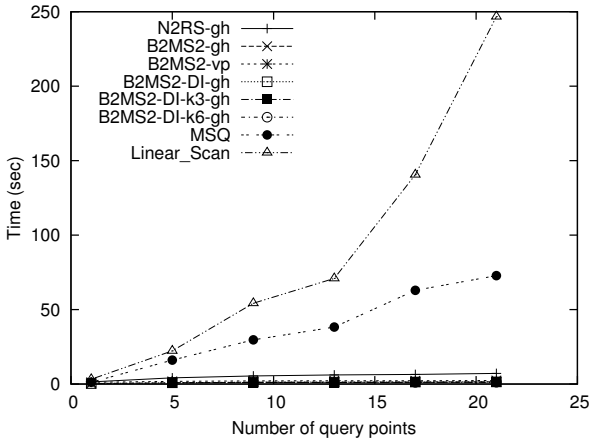
We consider three criteria when interpreting the results: running time, number of distance computations, and number of dominance tests. For the N^2RS and B^2MS^2 tests, tree construction is randomized, and query tuples are randomly selected, provided that they fall within a certain percentage of the dataset diameter of each other. This last constraint is used to simulate real-world queries. The dataset diameters of DNA, randomized-5d, uniform-20d, and dictionary are 11, 2, 3 and 30, respectively. We define a query fraction for each dataset, which is 30%, 40%, 45% and 10%, respectively. All randomly-selected query tuples of a dataset must fall within the range of the query fraction times dataset diameter.

Tree construction and query tuple selection for MSQ are slightly different. In its authors' implementation, trees are constructed by the M-Tree's BulkLoad method [10]. N Query tuples are selected by choosing a single random tuple and choosing its $N-1$ nearest neighbors as the other query tuples. To approximate this effect, when choosing query tuples for the N^2RS and B^2MS^2 implementations mentioned in the previous paragraph, we chose small query fractions.

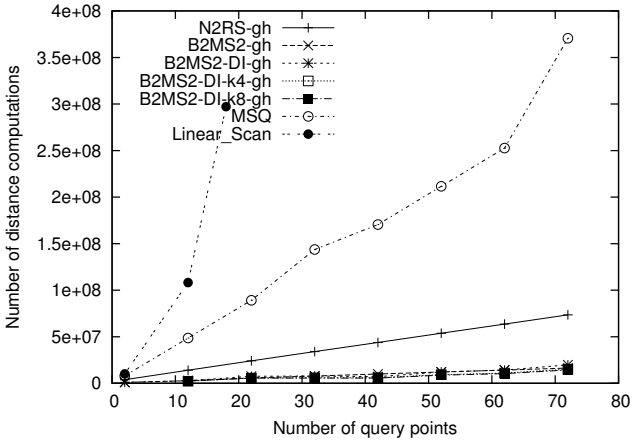
[3] Imitating the behavior of Mobios, we break the DNA sequence into overlapping, 12-character *fragments* and search among the fragments.



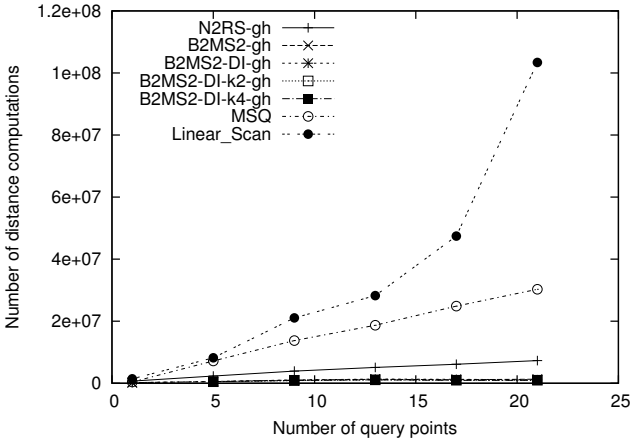
(a) DNA time



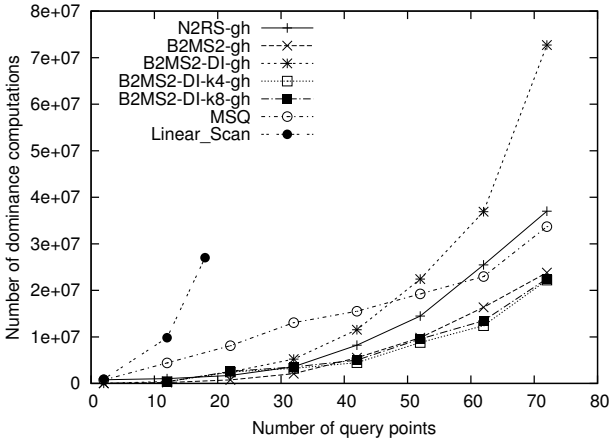
(b) dictionary time



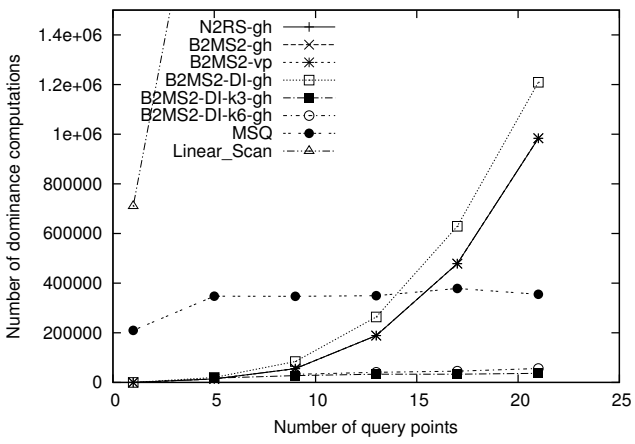
(c) DNA distance computations



(d) dictionary distance computations

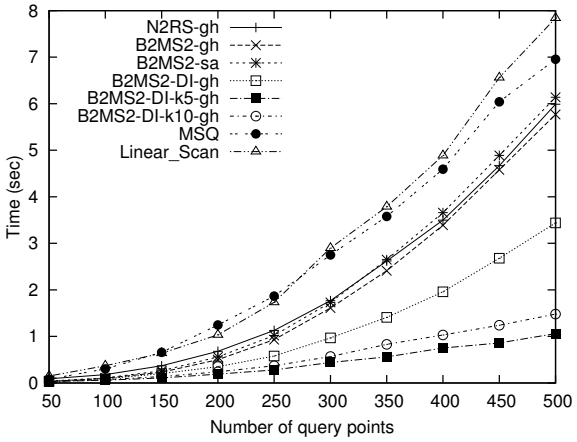


(e) DNA dominance tests

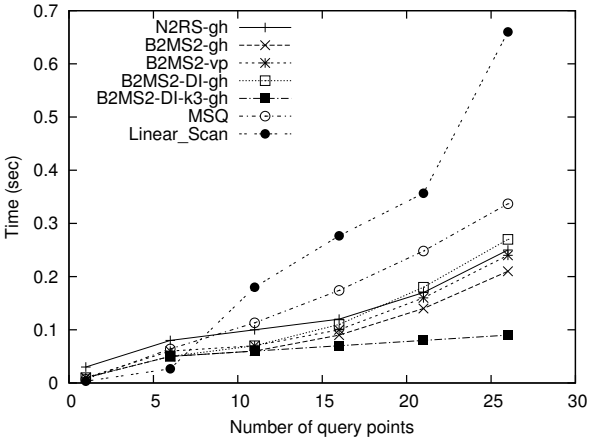


(f) dictionary dominance tests

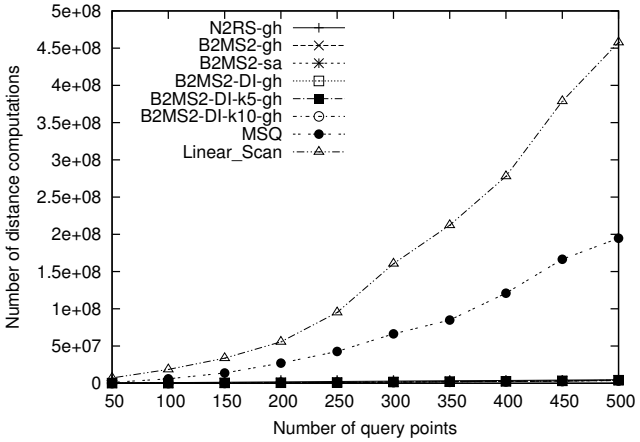
Figure 3. Experimental Results for DNA and dictionary



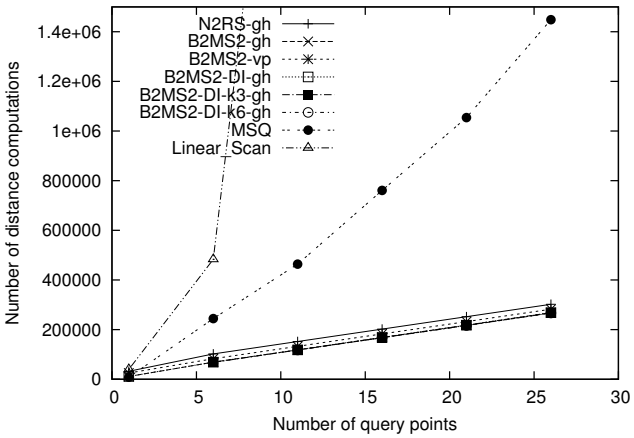
(a) random-5d time



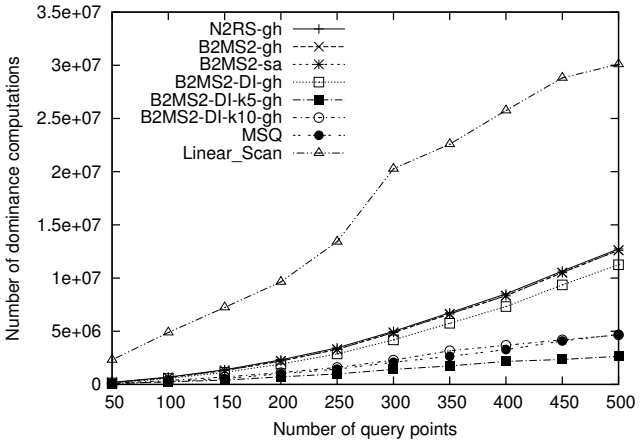
(b) uniform-20d time



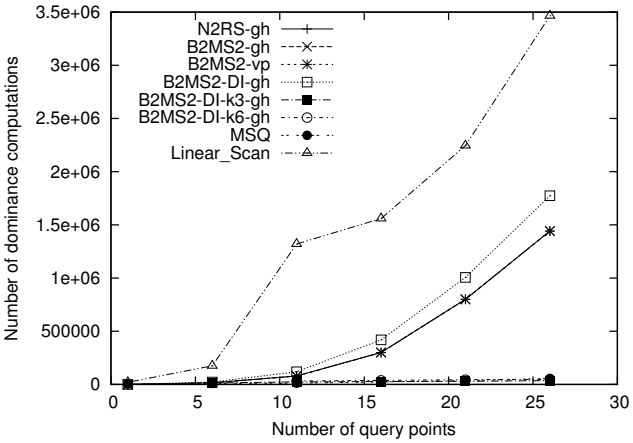
(c) random-5d distance computations



(d) uniform-20d distance computations



(e) random-5d dominance tests



(f) uniform-20d dominance tests

Figure 4. Experimental results for random-5d and uniform-20d

Table 1. Dataset Characteristics

Dataset	Data Type	Records	Type	Dist Metric	Branch Factor	Block Size
DNA	fragment	1000010	real	edit	40	10 KB
random-5d	vector (5d)	10000	synthetic	L_1 (Manhattan)	7	1 KB
uniform-20d	vector (20d)	10000	synthetic	L_2 (Euclidean)	20	10 KB
dictionary	string	354984	real	edit	45	10 KB

4.2 Testing Environment:

All tests were run on an AMD Opteron 2.0GHz machine with 2GB of main memory, running Linux (Fedora Core 4), with a 2.6.17 x86_64 kernel. Our algorithms were implemented in C++, using the STL for most data structures. The MSQ implementation was writeen by the authors of [8] in C++, using the M-Tree codebase for most data structures.

4.3 Performance of B^2MS^2 and N^2RS:

In this section we comment on the first two combinations tested: N^2RS-gh and B^2MS^2-gh (without dimensional indexing or k-materialization). Both use a GH-tree index, so the performance gains seen with B^2MS^2 are entirely due to the efficiency of the single-pass B^2MS^2 algorithm. (Plain) B^2MS^2-gh performs better than N^2RS-gh in every test. Overall, the performance improvement of B^2MS^2 over N^2RS is around a factor of 2.

4.4 Performance of MSQ

MSQ makes for some interesting comparisons. We would expect it to always perform similar to plain B^2RS^2-gh. However, while we see this expected behavior in figure 4(e), in other cases MSQ performs fewer dominance tests than some variants of B^2MS^2 as in figures 3(e) and 4(f). Often MSQ runs much slower as in figure 4(a).

Some of the differences can be explained by a difference in algorithm initialization. In our algorithm 2 on line 4, the B^2MS^2 ordered heap is initialized to the root node. In Figure 6 of [8] on line 3, however, the MSQ algorithm's heap is initialized to all children of the root node. This initialization difference can account for the apparent "startup cost" in dominance tests in figures 3(e) and 3(f), in which MSQ shows competitive performance only after the number of query tuples increases. Some of the more extreme performance differences are almost certainly grounded in implementation inefficiency or unoptimized performance cases, rather than in the MSQ algorithm itself. The MSQ implementation is capable of good performance, as evidenced by MSQ's very low number of dominance tests in figure 4(f). However, B^2MS^2 is typically much faster than MSQ.

4.5 The Benefit of Dynamic Indexes for Dominance Tests:

Here we note the benefit of the *dynamic indexing* optimization technique proposed in Section 3.1. Comparing the dynamically-indexed B^2MS^2-DI-gh to its nonindexed B^2MS^2-gh counterpart, we see that dynamic indexing improved performance noticeably for the random-5d dataset (figure 4(a)) but not for other datasets. In Figure 4(e) its ability to reduce dominance tests accounts for the substantial speedup in figure 4(a). In many other cases though, dynamic indexing is not effective when used alone. In these cases, the advantage of first comparing a candidate skyline point to (likely dominant) skyline points near their mutual closest query point, does not outweigh the cost of maintaining for every query point, a sorted bucket of close skyline points.

4.6 The Benefit of k-dispersion:

k-dispersion proves to be a flexible technique for reducing both distance computations and dominance tests. Every figure includes two k-dispersion values for comparison. In random-5d figures 4(a), 4(c), and 4(e) for example, we note that both the 5-dispersed B^2MS^2-DI-k5 and the 10-dispersed B^2MS^2-DI-k10 have relatively low values of k as compared to the number of query tuples $|Q|$ which can be as large as 500. Our tests showed that higher values of k cause performance to deteriorate, since more distance values are computed. In our experiments, a relatively small number of k can significantly improve the performance. In the figures, we can see that lower values of k (like $k = 5$) generally perform better than higher values (like $k = 10$) [4]. Overall, we can see that k-dispersion combined with dynamic indexing improved performance significantly for all datasets. The overall performance gain can be more than 300 times compared to N^2RS, and even more compared to a linear scan.

4.7 Skyline size:

Recent works such as [6] have suggested that for high-dimensional datasets, the set of skyline points becomes unmanageably large, given the increased difficulty of any one point dominating another. For the datasets we studied, using the constraints mentioned in Section 4.1, this appeared to be less of a problem in practice.

In our tests, skyline size scaled roughly linearly with increasing query tuple size. For the DNA dataset, with 12, 42, and 72 query tuples, the number of skyline points was 238, 3615, and 8219, respectively. For dictionary, 5, 30, and 45 query tuples gave 9, 313, and 750 skyline points. For random-5d, 50, 250, and 500 query tuples gave 166, 1884, and 3621 skyline points. And for uniform-20d, 21, 61, and 101 query tuples gave 614, 3019, and 4991 skyline points. Thus we believe our experiments model relatively useful skyline queries, rather than unrealistic queries which might return an intractably large (exponential) number of results.

5. Related Work

Skyline Queries: The Skyline Query has its origins in the problems of multi objective optimization [28], maximum vectors [16], convex hull [24], and contour generation [17]. It was first introduced in [2] and algorithms based on Block Nested Loop (BNL), Divide & Conquer, and B-Tree were presented. Further algorithms focused on improving BNL with sorting [9, 26] and bitmap

[4] The best choice for k is dependent on the dataset and the query. It can be determined experimentally by varying k for the same query. Estimating the best k in advance remains an open problem.

& sorted list index structures to return results progressively [29]. More recent work utilizes the nearest-neighbor type of search and R-tree to speedup the skyline query processing [15, 21]. However, until recently, all improved algorithms made use of (multi)dimensional index structures, rendering them unusable for efficiently answering MSS queries. Our work uses metric space index structures to efficiently compute the MSS.

A closely related work, the recently proposed B^2S^2 Spatial Skyline Query (SSQ) [27], is a stricter version of MSS query. Their approach uses geometric bounding structures, such as convex hull and Voronoi diagram, to reduce the search space. They demonstrate good results for planar graphs, but unfortunately their search structures preclude Spatial Skyline's applicability in moderate-to high-dimensional spaces as well as in general metric space. Very recently, Chen and Lian [8] developed a MSS algorithm MSQ for only the M-Tree metric space index. Our work computes the MSS while integrating two novel techniques to optimize distance computations and dominance tests. In addtion, we exploit various types of metric space indexes.

A separate line of research [23, 6, 7] shows how the results of a skyline query in high dimensional space can be further ranked and filtered based on characteristics of a point's frequency or dominance. Skyline cube algorithms [33] have been proposed which find skylines of subspaces. Our work, by contrast, focuses on solving the general skyline problem while making use of metric space indexes. In addition, we note that the *blowup* problem [6] for the large number of skyline points for the high dimensional space is less of a problem for typical queries in general metric spaces. In most of the cases, we expect the query points to be fairly close to each other as they generally share certain similarities.

Similarity Query in Metric Space: A variety of indexes have been proposed for metric space [30, 32, 5, 19, 3, 4, 11] to facilitate similarity queries. They generally target nearest neighbor (NN) , proximity / range, k-nearest neighbor (kNN), reverse k-nearest neighbor (rKNN) [32, 1], and aggregate nearest neighbor [22]. While these operations are closely related with the proposed metric-space query, they are incapable of retrieving the search results posed by such queries. A distinctive feature of the proposed MSS query is that it finds similar matches for a group of query points instead of a single match.

6. Conclusion

In this paper, we have introduced the skyline computation problem in metric space (MSS query). We have identified several key performance issues for processing such queries, including how to utilize existing metric space indices, and how to reduce the number of distance computations and dominance tests. We have developed two novel algorithms N^2RS and B^2MS^2 to answer MSS queries. In addition, we have proposed two new optimization techniques, dynamic indexing and k-dispersion. Our detailed experimental evaluation has shown our techniques can achieve a more than two orders of magnitude performance improvement.

7. Acknowledgments

We would like to thank Xiang Lian for providing us with the code used in [8].

8. REFERENCES

[1] Elke Achtert, Christian Böhm, Peer Kröger, Peter Kunath, Alexey Pryakhin, and Matthias Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *SIGMOD '06*, pages 515–526, 2006.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE '01*, pages 421–430, 2001.

[3] Tolga Bozkaya and Meral Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *SIGMOD '97*, pages 357–368, 1997.

[4] Tolga Bozkaya and Meral Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Trans. Database Syst.*, 24(3):361–404, 1999.

[5] Sergey Brin. Near neighbor search in large metric spaces. In *The VLDB Journal*, pages 574–584, 1995.

[6] Chee-Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and Zhenjie Zhang. Finding k-dominant skylines in high dimensional space. In *SIGMOD '06*, pages 503–514, 2006.

[7] Chee Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and Zhenjie Zhang. On high dimensional skylines. In *EDBT*, pages 478–495, 2006.

[8] Lei Chen and Xiang Lian. Dynamic skyline queries in metric spaces. In *EDBT '08*, 2008.

[9] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting, 2002.

[10] Paolo Ciaccia and Marco Patella. Bulk loading the m-tree. In *In 9th Australasian Database Conference (ADC'98*, pages 15–26, 1998.

[11] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree:an efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[12] E. Erkut. The discrete p-dispersion problem. *Eur. J. Opnl. Res.*, 46:48–60, 1990.

[13] David Fuhry, Ruoming Jin, and Donghui Zhang. Efficient skyline computation in metric space. Technical report, Kent State University, April 2008.

[14] D. Huttenlocher, D. Klanderman, and A. Rucklige. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, September 1993.

[15] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB 2002*, 2002.

[16] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[17] D. H. McLain. Drawing contours from arbitrary data points. *Comput. J.*, 17(4):318–324, 1974.

[18] Daniel Miranker, Weijia Xu, and Rui Mao. Mobios: A metric-space dbms to support biological discovery. *ssdbm*, 00:241, 2003.

[19] Gonzalo Navarro. Searching in metric spaces by spatial approximation. In *SPIRE/CRIWG*, pages 141–148, 1999.

[20] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, 48(3):443–53, 1970.

[21] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[22] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.*, 30(2):529–576, 2005.

[23] Jian Pei, Wen Jin, Martin Ester, and Yufei Tao. Catching the best views of skyline: a semantic approach based on decisive subspaces. In *VLDB '05*, pages 253–264, 2005.

[24] Franco P. Preparata and Michael I. Shamos. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[25] S. S. Ravi, D. J. Rosenkrantz, and G. K. Tayi. Heuristic and special case algorithms for dispersion problems. *Operations Research*, 42(2):299–310, 1994.

[26] Parke Godfrey Ryan. Maximal vector computation in large data sets.

[27] Mehdi Sharifzadeh and Cyrus Shahabi. The spatial skyline queries. In *VLDB'2006: Proceedings of the 32nd international conference on Very large data bases*, pages 751–762, 2006.

[28] R. Steuer. *Multiple criteria optimization*. Wiley, New York, NY, 1986.

[29] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. Efficient progressive skyline computation. In *VLDB 2001*, pages 301–310, 2001.

[30] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[31] Grady Ward. *Moby Word Lists*. Project Gutenberg, 2002.

[32] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA '93*, pages 311–321, 1993.

[33] Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang. Efficient computation of the skyline cube. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 241–252. VLDB Endowment, 2005.