On Finding Skylines in External Memory

Cheng Sheng
CUHK
Hong Kong
csheng@cse.cuhk.edu.hk

Yufei Tao
CUHK
Hong Kong
taoyf@cse.cuhk.edu.hk

ABSTRACT

We consider the *skyline problem* (a.k.a. the *maxima problem*), which has been extensively studied in the database community. The input is a set P of d-dimensional points. A point *dominates* another if the former has a lower coordinate than the latter on every dimension. The goal is to find the *skyline*, which is the set of points $p \in P$ such that p is not dominated by any other data point. In the external-memory model, the 2-d version of the problem is known to be solvable in $O((N/B)\log_{M/B}(N/B))$ I/Os, where N is the cardinality of P, B the size of a disk block, and M the capacity of main memory. For fixed $d \geq 3$, we present an algorithm with I/O-complexity $O((N/B)\log^{d-2}_{M/B}(N/B))$. Previously, the best solution was adapted from an in-memory algorithm, and requires $O((N/B)\log_2^{d-2}(N/M))$ I/Os.

Categories and Subject Descriptors

F2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems—geometric problems and computations

General Terms

Algorithms, theory

Keywords

Skyline, admission point, pareto set, maxima set

1. INTRODUCTION

This paper studies the *skyline problem*. The input is a set P of d-dimensional points in *general position*, i.e., no two points of P share the same coordinate on any dimension. Given a point $p \in \mathbb{R}^d$, denote its i-th ($1 \leq i \leq d$) coordinate as $p[i]$. A point p_1 is said to *dominate* another point p_2, represented as $p_1 \prec p_2$, if p_1 has a smaller coordinate on all d dimensions, namely:

$$\forall i = 1, ..., d,\ p_1[i] < p_2[i].$$

The goal is to compute the *skyline* of P, denoted as $SKY(P)$, which includes all the points in P that are not dominated by any other point, namely:

$$SKY(P) = \{p \in P \mid \nexists p' \in P \text{ s.t. } p' \prec p\}. \quad (1)$$

The skyline is also known under other names such as the *pareto set*, the set of *admission points*, and the set of *maximal vectors* (see [24]).

Figure 1: The skyline is $\{1, 5, 7\}$

Ever since its debut in the database literature a decade ago [7], skyline computation has generated considerable interests in the database area (see [24] for a brief survey). This is, at least in part, due to the relevance of skylines to *multi-criteria optimization*. Consider, for example, a hotel recommendation scenario, where each hotel has two attributes `price` and `rating` (a smaller rating means better quality). Figure 1 illustrates an example with 8 hotels, of which the skyline is $\{1, 5, 7\}$. Every hotel *not* in the skyline is worse than at least one hotel in the skyline on both dimensions, i.e., more expensive and rated worse at the same time. In general, for any scoring function that is monotone on all dimensions, the skyline always contains the *best* (top-1) point that minimizes the function. This property is useful when it is difficult, if not impossible, for a user to specify a suitable scoring function that accurately reflects her/his preferences on the relative importance of various dimensions. In Figure 1, for instance, $\{1, 5, 7\}$ definitely includes the best

PODS'11, June 13–15, 2011, Athens, Greece.

hotel, no matter the user emphasizes more, and how much more, on `price` or `rating`.

1.1 Computation model

Our complexity analysis is under the standard *external memory* (*EM*) model [2], which has been successful in capturing the characteristics of algorithms dealing with massive data that do not fit in memory (see [27] for a broad summary of results in this model). Specifically, in this model, a computer has a main memory that is able to accommodate M words, and a disk of unbounded size. The disk is formatted into disjoint *blocks*, each of which contains B consecutive words. The memory should have at least two blocks, i.e., $M \geq 2B$. An *I/O operation* reads a block of data from the disk into memory, or conversely, writes a block of memory information into the disk. The time complexity is measured in the number of I/Os performed by an algorithm. CPU calculation is free.

In EM, *linear* cost should be interpreted as $O(N/B)$ for a dataset of size N, as opposed to $O(N)$ in a memory-resident model such as RAM. In this paper, *poly-logarithmic* should be understood as $O(\text{polylog}_{M/B}(N/B))$, instead of $O(\text{polylog}\, N)$, namely, it is important to achieve base M/B. In this paper, a function is said to be *near-linear* if it is in $O((N/B)\,\text{polylog}_{M/B}(N/B))$ but not in $O(N/B)$.

1.2 Previous results

In internal memory, Matousek [21] showed how to find the skyline in $O(N^{2.688})$ time when the dimensionality d of the dataset P is as high as the number N of points in P. In 2-d space, Kung et al. [19] proposed an $O(N \log N)$-time algorithm. For any fixed dimensionality $d \geq 3$, they also gave an algorithm with time complexity $O(N \log^{d-2} N)$. Bentley [4] developed an alternative algorithm achieving the same bounds as those of [19]. Kirkpatrick and Seidel [16] presented algorithms whose running time is sensitive to the result size, and has the same complexity as the algorithms in [4, 19] when the skyline has $\Omega(N)$ points. It can be shown that any algorithm in the *comparison class*[1] must incur $\Omega(N \log N)$ time, implying that the solutions of [4, 19] are already optimal in this class for $d = 2$ and 3 (see also some recent results on instance optimality due to Afshani et al. [1]). For $d \geq 4$, Gabow et al. [11] discovered an algorithm terminating in $O(N \log^{d-3} N \log\log N)$ time, which still remains the best result up to this day. Note, however, that the solution of [11] does not belong to the comparison class, due to its reliance on the van Emde Boas structure [26] that uses features of the RAM model. Faster algorithms have been developed in some special circumstances where, for example, the data follow special distributions [5, 6, 10], or each dimension has a small domain [22].

All the RAM algorithms can be applied in the EM model directly by treating the disk as virtual memory. Such a brute-force approach, however, can be expensive in practice because it fails to take into account the effects of blocking, which do not exist in RAM but are inherent in external memory. For example, running the solution of [11] in EM naively would entail $O(N \log^{d-3} N \log\log N)$ I/Os, which amounts to reading the entire dataset $O(B \log^{d-3} N \log\log N)$ times (B is at the order of thousands in practice). Hence, there is a genuine need to design I/O-oriented algorithms. For $d = 2$, such an algorithm can be easily found, as Kung et al. [19] showed that the problem can be settled by sorting the data followed by a single scan (we will come back to this in Section 2), which takes $O((N/B)\log_{M/B}(N/B))$ I/Os in total. To our knowledge, for general d, the RAM algorithm that can be most efficiently adapted to EM is the one by Bentley [4], which performs $O((N/B)\log_2^{d-2}(N/M))$ I/Os – note that the base of the log is 2, instead of M/B. We are not aware of any algorithm that can achieve near-linear (or better) running time.

For $d \geq 3$, the skyline of a dataset P can be trivially obtained by computing the cartesian product $P \times P$ (i.e., by comparing all pairs of points in P) which, in turn, can be produced by a *blocked nested loop (BNL)* in $\Theta(N^2/(MB))$ I/Os. It has been observed [7] that such a quadratic complexity is too slow in practice for large N. In the past decade, several algorithms, as we survey below, have been designed to alleviate the deficiency, typically by leveraging the *transitivity* of the dominance relation (i.e., $p_1 \prec p_2$ and $p_2 \prec p_3$ imply $p_1 \prec p_3$). Although empirical evaluation has confirmed their effectiveness on selected datasets, *none* of those algorithms has been proved to be asymptotically faster than *BNL* in the worst case. We say that they are captured by the *quadratic trap*.

Borzsonyi et al. [7] presented a *divide and conquer (DC)* method that partitions P into disjoint groups $P_1, ..., P_s$ where the number s of groups is large enough so that each P_i ($i \leq s$) fits in memory. *DC* proceeds by invoking an in-memory algorithm to find the skyline $SKY(P_i)$ of each P_i, and then, deleting those points of $SKY(P_i)$ dominated by some point in the skyline of another group. Although divide and conquer is a promising paradigm for attacking the skyline problem (it is also employed in our solutions), its application in *DC* is heuristic and does not lead to any interesting performance bound.

The *sort first skyline (SFS)* algorithm by Chomicki et al. [9] works by sorting the points $p \in P$ in ascending order of $score(p)$, where *score* can be any function $\mathbb{R}^d \to \mathbb{R}$ that is monotonically increasing on all dimensions. The monotonicity ensures that, $p_1 \prec p_2$ implies $score(p_1) < score(p_2)$ (but the opposite is *not* true). As a result, a point $p \in P$ cannot be dominated by any point that ranks *behind* it in the ordering. Following this rationale, *SFS* scans P in its sorted order, and maintains the skyline Σ of the points already seen so far (note that $\Sigma \subseteq SKY(P)$ at any time). As expected, the choice of *score* is crucial to the efficiency of the algorithm. No choice, unfortunately, is known to be able to escape the quadratic trap in the worst case.

In *SFS*, sorting is carried out with the standard external sort. Intuitively, if mutual comparisons are carried out among the data that ever co-exist in memory (during the external sort), many points may get discarded right away once confirmed to be dominated, at no extra I/O cost at all. Based on this idea, Godfrey et al. [12] developed the *linear elimination sort for skyline (LESS)* algorithm. *LESS* has the property that, it terminates in linear expected I/Os under the *independent-and-uniform* assumption (i.e., all di-

[1]A skyline algorithm is *comparison-based* if it can infer the dominance relation by only comparing pairs of points. The comparison class includes all such algorithms.

method	I/O complexity	remark
Kung et al. [19]	$O(N \log^{d-2} N)$	
Gabow et al. [11]	$O(N \log^{d-3} N \log\log N)$	not in the comparison class.
Bentley [4]	$O((N/B) \log_2^{d-2}(N/M))$	adapted from Bentley's $O(N \log^{d-2} N)$ algorithm in RAM
BNL	$\Theta(N^2/(MB))$	also applies to the *BNL* variant of Borzsonyi et al. [7]
DC [7]	$\Omega(N^2/(MB))$	
SFS [9]	$O(N^2/(MB))$	
LESS [12]	$O(N^2/(MB))$	
RAND [24]	$O(\mu N/(MB))$ expected	μ is the number of points in the skyline, which can be $\Omega(N)$.
this paper	$\boldsymbol{O((N/B) \log_{M/B}^{d-2}(N/B))}$	**optimal for $d = 3$ in the comparison class**

Table 1: Comparison of skyline algorithms for fixed $d \geq 3$

mensions are independent, and the points of P distribute uniformly in the data space), provided that the memory size M is not too small [12]. When the assumption does not hold, however, it remains unknown whether the cost of *LESS* is $o(N^2/(MB))$.

Sarma et al. [24] described an output-sensitive randomized algorithm *RAND*, which continuously shrinks P with repetitive iterations, each of which performs a three-time scan on the current P as follows. The first scan takes a random sample set $S \subseteq P$ with size $\Theta(M)$. The second pass replaces (if possible) some samples in S with other points that have stronger pruning power. All samples at the end of this scan are guaranteed to be in the skyline, and thus removed from P. The last scan further reduces $|P|$, by eliminating all the points that are dominated by some sample. At this point, another iteration sets off as long as $P \neq \emptyset$. *RAND* is efficient when the result has a small size. Specifically, if the skyline has μ points, *RAND* entails $O(\mu N/(MB))$ I/Os in expectation. When $\mu = \Omega(N)$, however, the time complexity falls back in the quadratic trap.

There is another line of research that concerns preprocessing a dataset P into a structure that supports fast retrieval of the skyline, as well as insertions and deletions on P (see [14, 15, 18, 20, 23] and the references therein). In our context, such pre-computation-based methods do not have a notable advantage over the algorithms mentioned earlier.

1.3 Our results

Our main result is:

THEOREM 1.1. *The skyline of N points in $\mathbb{R}^d$ can be computed in $O((N/B) \log_{M/B}^{d-2}(N/B))$ I/Os for any fixed $d \geq 3$.*

The theorem concerns only $d \geq 3$ because, as mentioned before, the skyline problem is known to be solvable in $O((N/B) \log_{M/B}(N/B))$ I/Os in 2-d space. Unlike the result of Godfrey et al. [12], we make no assumption on the data distribution. Our algorithm is the first one that beats the quadratic trap and, at the same time, achieves near-linear running time. In 3-d space, our I/O complexity $O((N/B) \log_{M/B}(N/B))$ is asymptotically optimal in the class of comparison-based algorithms. For any fixed d, Theorem 1.1 shows that the skyline problem can be settled in $O(N/B)$ I/Os, when $N/B = (M/B)^c$ for some constant c (a situation that is likely to happen in practice). No previous algorithm is known to have such a property. See Table 1 for a comparison of our and existing results.

The core of our technique is a *distribution-sweep*[2] algorithm for solving the *skyline merge* problem, where we are given s skylines $\Sigma_1, ..., \Sigma_s$ that are separated by $s-1$ hyperplanes orthogonal to a dimension; and the goal is to return the skyline of the union of all the skylines, namely, $SKY(\Sigma_1 \cup ... \cup \Sigma_s)$. It is not hard to imagine that this problem lies at the heart of computing the skyline using a divide-and-conquer approach. Indeed, the lack of a fast solution to skyline merging has been the obstacle in breaking the curse of quadratic trap, as can be seen from the divide-and-conquer attempt of Borzsonyi et al. [7]. We overcome the obstacle by lowering the dimensionality to 3 gradually, and then settling the resulting 3-d problem in linear I/Os. Our solution can also be regarded as the counterpart of Bentley's algorithm [4] in external memory.

Remark. We have defined our problem by assuming that P is in general position. The skyline $SKY(P)$ is still well defined without this assumption. Specifically, let P be a set of points in $\mathbb{R}^d$ (implying that P has no duplicates). Given two points p, p' in P, we have $p \prec p'$ if $p[i] \leq p'[i]$ for all $1 \leq i \leq d$. Note that since $p \neq p'$, the equality does not hold for at least one i. Then, $SKY(P)$ is still given by Equation 1. Our algorithms can be extended to solve this version of the skyline problem, retaining the same performance guarantee as in Theorem 1.1. Details can be found in Section 3.3.

2. PRELIMINARIES

In this section, we review some skyline algorithms designed for memory-resident data. The purposes of the review are three fold. First, we will show that the 2-d solution of Kung et al. [19] can be easily adapted to work in the EM model. Second, our discussion of their algorithm and Bentley's algorithm [4] for $d \geq 3$ not only clarifies some characteristics of the underlying problems, but also sheds light on some obstacles preventing a direct extension to achieve near-linear time complexity in external memory. Finally, we briefly explain the cost lower bound established in [19] and why a similar bound also holds in the I/O context.

Let us first agree on some terminologies. We refer to the

[2]An algorithm paradigm proposed by Goodrich et al. [13] that can be regarded as the counterpart of plane-sweep in external memory.

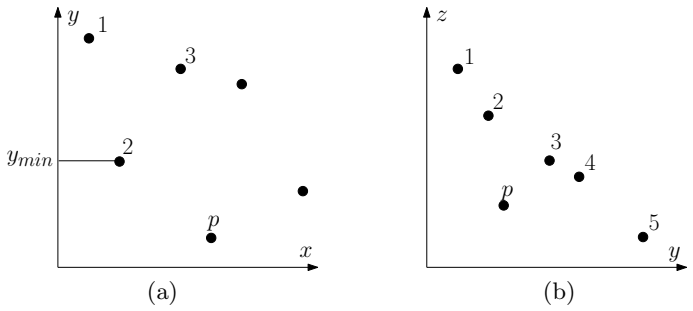

Figure 2: Illustration of algorithms by Kung et al. [19]: (a) 2-d, (b) 3-d

first, second, and third coordinate of a point as its x-, y-, and z-coordinate, respectively. Sometimes, it will be convenient to extend the definition of dominance to subspaces in a natural manner. For example, in case p_1 has smaller x- and y-coordinates than p_2, we say that p_1 dominates p_2 in the x-y plane. No ambiguity can arise as long as the subspace concerning the dominance is always mentioned.

2-d. The skyline $SKY(P)$ of a set P of 2-d points can be extracted by a single scan, provided that the points of P have been sorted in ascending order of their x-coordinates. To understand the rationale, consider any point $p \in P$; and let P' be the set of points of P that rank before p in the sorted order. Apparently, p cannot be dominated by any point that ranks *after* p, because p has a smaller x-coordinate than any of those points. On the other hand, p is dominated by *some* point in P' if and only if the y-coordinate of p is greater than y_{min}, where y_{min} is the smallest y-coordinate of all the points in P'. See Figure 2a where P' includes points 1, 2, 3; and that no point in P' dominates p can be inferred from the fact that p has a lower y-coordinate than y_{min}. Therefore, to find $SKY(P)$, it suffices to read P in its sorted order, and at any time, keep the smallest y-coordinate y_{min} of all the points already seen. The next point p scanned is added to $SKY(P)$ if its y-coordinate is below y_{min}, in which case y_{min} is updated accordingly. In the EM model, this algorithm performs $O((N/B)\log_{M/B}(N/B))$ I/Os, which is the time complexity of sorting N elements in external memory.

3-d. Suppose that we have sorted P in ascending order of their x-coordinates. Similar to the 2-d case, consider any point $p \in P$, with P' being the set of points before p. It is clear that p cannot be dominated by any point that ranks after p. Judging whether p is dominated by a point of P', however, is more complex than the 2-d scenario. The general observation is that, since all points of P' have smaller x-coordinates than p, we only have to check whether p is dominated by some point of P' in the y-z plane. Imagine that we project all the points of P' onto the y-z plane, which yields a 2-d point set P''. Let Σ be the (2d) skyline of P''. It is sufficient to decide whether a point in Σ dominates p in the y-z plane.

It turns out that such a dominance check can be done efficiently. In general, a 2-d skyline is a "staircase". In the y-z plane, if we walk along the skyline in the direction of growing y-coordinates, the points encountered must have descending z-coordinates. Figure 2b illustrates this with a Σ that consists of points 1, ..., 5. To find out whether p is dominated by any point of Σ in the y-z plane, we only need to find the predecessor o of p along the y-dimension among the points of Σ, and give a "yes" answer if and only if o has a lower z-coordinate than p. In Figure 2b, the answer is "no" because the predecessor of p, i.e., point 2, actually has a greater z-coordinate than p. Returning to the earlier context with P', a "no" indicates that p is not dominated by any point in P', and therefore, p belongs to the skyline $SKY(P)$.

Based on the above reasoning, the algorithm of [19] maintains Σ while scanning P in its sorted order. To find predecessors quickly, the points of Σ are indexed by a binary tree T on their y-coordinates. The next point p is added to $SKY(P)$ upon a "no" answer as explained before, which takes $O(\log N)$ time with the aid of T. Furthermore, a "no" also necessitates the deletion from Σ of all the points that are dominated by p in the y-z plane (e.g., points 3, 4 in Figure 2b). Using T, this requires only $O(\log N)$ time per point removed. As each point of P is deleted at most once, the entire algorithm finishes in $O(N \log N)$ time.

A straightforward attempt of externalizing the algorithm is to implement T as a B-tree. This will result in the total execution time of $\Theta(N \log_B N)$, which is higher than the cost $O((N/B)\log_{M/B}(N/B))$ of our solution by a factor of $\Omega(B \log_B M)$. The deficiency is due to the fact (see [13]) that plane sweep, which is the methodology behind the above algorithm, is ill-fitted in external memory, because it issues a large number of queries (often $\Omega(N)$), rendering it difficult to control the overall cost to be at the order of N/B.

Following a different rationale, Bentley [4] gave another algorithm of $O(N \log N)$ time. We will not elaborate his solution here because our algorithm in the next section degenerates into Bentley's, when M and B are set to constants satisfying $M/B = 2$.

Dimensionalities at least 4. Kung et al. [19] and Bentley [4] deal with a general d-dimensional ($d \geq 4$) dataset P by divide-and-conquer. More specifically, their algorithms divide P into P_1 and P_2 of roughly the same size by a hyperplane perpendicular to dimension 1. Assume that the points of P_1 have smaller coordinates on dimension 1 than those of P_2. Let Σ_1 be the skyline $SKY(P_1)$ of P_1, and similarly, $\Sigma_2 = SKY(P_2)$. All points of Σ_1 immediately belong to $SKY(P)$, but a point $p \in \Sigma_2$ is in $SKY(P)$ if and only if no point in Σ_1 dominates p. Hence, after obtaining Σ_1 and Σ_2 from recursion, a skyline merge is carried out to evict such points as p.

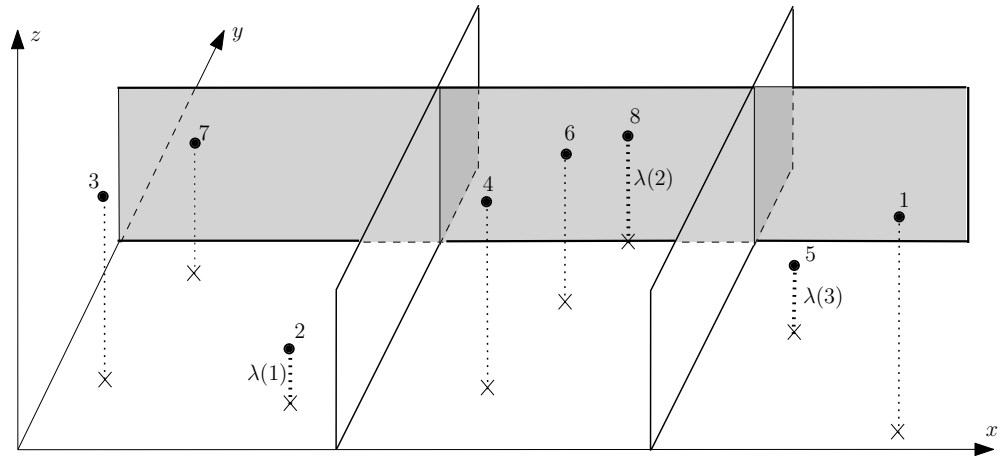

Figure 3: Illustration of 3-d skyline merge. The value of s is 3. Only the points already encountered are shown. Points are labeled in ascending order of their y-coordinates (which is also the order they are fetched). Point 8 is the last one seen. Each cross is the projection of a point in the x-y plane. $\Sigma(1)$ contains points 2, 3, 7, $\Sigma(2)$ includes 4, 6, 8, and $\Sigma(3)$ has 5, 1. $\lambda(1), \lambda(2), \lambda(3)$ equal the z-coordinate of point 2, 8, 5, respectively. Point 8 does not belong to $SKY(P)$ because its z-coordinate is larger than $\lambda(1)$ (i.e., it violates Inequality 2 on $j = 1$).

Externalization of the algorithms of Kung et al. [19] and Bentley [4] is made difficult by a common obstacle. That is, the partitioning in the divide-and-conquer is binary, causing a recursion depth of $\Omega(\text{polylog}(N/M))$. To obtain the performance claimed in Theorem 1.1, we must limit the depth to $O(\text{polylog}_{M/B}(N/B))$. This cannot be achieved by simply dividing P into a greater number $s > 2$ of partitions $P_1, ..., P_s$, because doing so may compromise the efficiency of merging skylines. To illustrate, let $\Sigma_i = SKY(P_i)$ for each $1 \leq i \leq s$. A point $p \in S_i$ must be compared to $SKY(P_j)$ for all $j < i$. Applying the skyline merge strategy of [4] or [19] would blow up the cost by a factor of $\Omega(s^2)$, which would offset all the gains of a large s. Remedying the drawback calls for a new skyline merge algorithm, which we give in the next section.

Cost lower bound. Kung et al. [19] proved that any 2-d skyline algorithm in the comparison class must incur $\Omega(N \log N)$ execution time. To describe the core of their argument, let us define the *rank permutation* of a sequence S of distinct numbers $(x_1, ..., x_N)$, as the sequence $(r_1, ..., r_N)$ where r_i $(1 \leq i \leq N)$ is the number of values of S that are at most x_i. For example, the rank permutation of $(9, 20, 3)$ is $(2, 3, 1)$. Kung et al. [19] identified a series of hard datasets, where each dataset P has N points $p_1, ..., p_N$ whose x-coordinates can be any integers. They showed that, any algorithm that correctly finds the skyline of P must have determined the rank permutation of the sequence formed by the x-coordinates of $p_1, ..., p_N$. In the EM model, it is known [2] that deciding the rank permutation of a set of N integers demands $\Omega((N/B) \log_{M/B}(N/B))$ I/Os in the worst case for any comparison-based algorithm. It thus follows that this is also a lower bound for computing 2-d skylines in external memory. Note that the same bound also holds in higher-dimensional space where the problem is no easier than in the 2-d space.

It is worth mentioning that the I/O lower bound $\Omega((N/B) \log_{M/B}(N/B))$ is also a direct corollary of a result due to Arge et al. [3].

3. OUR SKYLINE ALGORITHM

We will present the proposed solution in a step-by-step manner. Section 3.1 first explains the overall divide-and-conquer framework underpinning the algorithm by clarifying how it works in 3-d space. To tackle higher dimensionalities d, there is another layer of divide-and-conquer inside the framework, as elaborated in Section 3.2 for $d = 4$. The 4-d description of our algorithm can then be easily extended to general d, which is the topic of Section 3.3.

3.1 3-d

Our algorithm accepts as input a dataset P whose points are sorted in ascending order of x-coordinates. If the size N of P is at most M (i.e., the memory capacity), we simply find the skyline of P using a memory-resident algorithm. The I/O cost incurred is $O(N/B)$.

In case $N > M$, we divide P into $s = \Theta(M/B)$ partitions $P(1), ..., P(s)$ with roughly the same size, such that each point in $P(i)$ has a smaller x-coordinate than all points in $P(j)$ for any $1 \leq i < j \leq s$. As P is already sorted on the x-dimension, the partitioning can be done in linear cost, while leaving the points of each $P(i)$ sorted in the same way.

The next step is to obtain the skyline $\Sigma(i)$ of each $P(i)$, i.e., $\Sigma(i) = SKY(P(i))$. Since this is identical to solving the original problem (only on a smaller dataset), we recursively invoke our algorithm on $P(i)$. Now consider the moment when all $\Sigma(i)$ have been returned from recursion. Our algorithm proceeds by performing a *skyline merge*, which finds the skyline of the union of all $\Sigma(i)$, that is, $SKY(\Sigma(1) \cup ... \cup \Sigma(s))$, which is exactly $SKY(P)$. We enforce an invariant that, $SKY(P)$ be returned in a disk file where the points are sorted in ascending order of y-

coordinates (to be used by the upper level of recursion, if any). Due to recursion, the invariant implies that, the same ordering has been applied to all the $\Sigma(i)$ at hand.

We now elaborate the details of the skyline merge. $SKY(P)$ is empty in the outset. $\Sigma(1), ..., \Sigma(s)$ are scanned synchronously in ascending order of y-coordinates. In other words, the next point fetched is guaranteed to have the lowest y-coordinate among the points of all $\Sigma(i)$ that have not been encountered yet. As $s = \Theta(M/B)$, the synchronization can be achieved by assigning a block of memory as the input buffer to each $\Sigma(i)$. We maintain a value $\lambda(i)$, which equals the minimum z-coordinate of all the points that have already been seen from $\Sigma(i)$. If no point from $\Sigma(i)$ has been read, $\lambda(i) = \infty$.

We decide whether to include a point p in $SKY(P)$ when p is fetched by the synchronous scan. Suppose that p is from $\Sigma(i)$ for some i. We add p to $SKY(P)$ if

$$p[3] < \lambda(j), \forall j < i \tag{2}$$

where $p[3]$ denotes the z-coordinate of p. See Figure 3 for an illustration. The lemma below shows the correctness of this rule.

LEMMA 3.1. *$p \in SKY(P)$ if and only if Inequality 2 holds.*

PROOF. Clearly, p cannot be dominated by any point in $\Sigma(i+1), ..., \Sigma(s)$ because p has a smaller x-coordinate than all those points. Let S be the set of points in $\Sigma(j)$ already scanned before p, for any $j < i$. No point $p' \in \Sigma(j) \setminus S$ can possibly dominate p, as p has a lower y-coordinate than p'. On the other hand, all points in S dominate p in the x-y plane. Thus, *some* point in S dominates p in $\mathbb{R}^3$ if and only if Inequality 2 holds. □

We complete the algorithm description with a note that a single memory block can be used as an output buffer, so that the points of $SKY(P)$ can be written to the disk in linear I/Os, by the same order they entered $SKY(P)$, namely, in ascending order of their y-coordinates. Overall, the skyline merge finishes in $O(N/B)$ I/Os.

Running time. Denote by $F(N)$ the I/O cost of our algorithm on a dataset with cardinality N. It is clear from the above discussion that

$$F(N) = \begin{cases} O(N/B) & \text{if } N \leq M \\ s \cdot F(N/s) + G(N) & \text{otherwise} \end{cases} \tag{3}$$

where $G(N) = O(N/B)$ is the cost of a skyline merge. Solving the recurrence gives $F(N) = O((N/B)\log_{M/B}(N/B))$.

3.2 4-d

To find the skyline of a 4-d dataset P, we proceed as in the 3-d algorithm by using a possibly smaller $s = \Theta(\min\{\sqrt{M}, M/B\})$. The only difference lies in the way that a skyline merge is performed. Formally, the problem we face in a skyline merge can be described as follows. Consider a partition $P(1), ..., P(s)$ of P such that each point in $P(i)$ has a smaller coordinate on dimension 1 than all points in $P(j)$, for any $1 \leq i < j \leq s$. Equivalently, the data space is divided into s *slabs* $\sigma_1(1), ..., \sigma_1(s)$ by $s-1$ hyper-planes orthogonal to dimension 1 such that $P(i) = P \cap \sigma_1(i)$ for all $1 \leq i \leq s$. We are given the skyline $\Sigma_1(i)$ of each $P(i)$, where the points of $\Sigma_1(i)$ are sorted in ascending order of their 2nd coordinates. The goal is to compute $SKY(\Sigma_1(1) \cup ... \cup \Sigma_1(s))$, which is equivalent to $SKY(P)$. Further, the output order is important (for backtracking to the upper level of recursion): we want the points of $SKY(P)$ to be returned in ascending order of their 2nd coordinates.

The previous subsection solved the problem in 3-d space with $O(N/B)$ I/Os where $N = |P|$. In 4-d space, our objective is to pay only an extra factor of $O(\log_{M/B}(N/B))$ in the cost. We fulfill the purpose with an algorithm called `preMerge-4d`, the input of which includes

- slabs $\sigma_1(1), ..., \sigma_1(s)$
- a set Π of points sorted in ascending order of their 2nd coordinates. Π has the property that, if two points $p_1, p_2 \in \Pi$ fall in the same slab, they do not dominate each other.

`preMerge-4d` returns the points of $SKY(\Pi)$ in ascending order of their *3rd* coordinates.

Although stated somewhat differently, the problem settled by `preMerge-4d` is (almost) the same as skyline merge. Notice that Π can be obtained by merging $\Sigma_1(1), ..., \Sigma_1(s)$ in $O(N/B)$ I/Os. Moreover, we can sort the points of $SKY(\Pi)$ (output by `preMerge-4d`) ascendingly on dimension 2 to fulfill the order requirement of skyline merge, which demands another $O((N/B)\log_{M/B}(N/B))$ I/Os.

Algorithm preMerge-4d. In case Π has at most M points, `preMerge-4d` solves the problem in memory. Otherwise, in $O(|\Pi|/B)$ I/Os the algorithm divides Π into s partitions $\Pi(1), ..., \Pi(s)$ of roughly the same size, with the points of $\Pi(i)$ having smaller 2nd coordinates than those of $\Pi(j)$ for any $1 \leq i < j \leq s$. We then invoke `preMerge-4d` recursively on each $\Pi(i)$, feeding the same $\{\sigma_1(1), ..., \sigma_1(s)\}$, to calculate $\Sigma_2(i) = SKY(\Pi(i))$. Apparently, $SKY(\Pi)$ is equivalent to the skyline of the union of all $\Sigma_2(i)$, namely, $SKY(\Pi) = SKY(\Sigma_2(1) \cup ... \cup \Sigma_2(s))$. It may appear that we are back to where we started — this is another skyline merge! The crucial difference, however, is that only two dimensions remain "unprocessed" (i.e., dimensions 3 and 4). In this case, the problem can be solved directly in linear I/Os, by a synchronous scan similar to the one in Section 3.1.

By recursion, the points of each $\Sigma_2(i)$ have been sorted ascendingly on dimension 3. This allows us to enumerate the points of $\Sigma_2(1) \cup ... \cup \Sigma_2(s)$ in ascending order of their 3rd coordinates, by synchronously reading the $\Sigma_2(i)$ of all $i \in [1, s]$. In the meantime, we keep track of s^2 values $\lambda(i_1, i_2)$ for every pair of $i_1, i_2 \in [1, s]$. Specifically, $\lambda(i_1, i_2)$ equals the lowest 4th coordinate of all the points in $\sigma_1(i_1) \cap \Sigma_2(i_2)$ that have been scanned so far; or $\lambda(i_1, i_2) = \infty$ if no such point exists. Note that the choice of s makes it possible to maintain all $\lambda(i_1, i_2)$ in memory, and meanwhile, allocate an input buffer to each $\Sigma_2(i)$ so that the synchronous scan can be completed in linear I/Os.

$SKY(\Pi)$ is empty at the beginning of the synchronous scan. Let p be the next point fetched. Suppose that p falls

in $\sigma_1(i_1)$, and is from $\Sigma_2(i_2)$, for some i_1, i_2. We insert p in $SKY(\Pi)$ if

$$p[4] < \lambda(j_1, j_2), \forall j_1 < i_1, j_2 < i_2 \quad (4)$$

where $p[4]$ is the coordinate of p on dimension 4. An argument similar to the proof of Lemma 3.1 shows that $p \in SKY(\Pi)$ if and only if the above inequality holds. Note that checking the inequality happens in memory, and incurs no I/O cost. Finally, as the points of $SKY(\Pi)$ enter $SKY(\Pi)$ in ascending order of their 3rd coordinates, they can be written to the disk in the same order.

Running time. Let $H(K)$ be the I/O cost of `preMerge-4d` when Π has K points. We have

$$H(K) = \begin{cases} O(K/B) & \text{if } K \leq M \\ s \cdot H(K/s) + O(K/B) & \text{otherwise} \end{cases}$$

where $s = \Omega(\sqrt{M/B})$. This recurrence gives $H(K) = O((K/B)\log_{M/B}(K/B))$.

Following the notations in Section 3.1, denote by $G(N)$ the cost of a skyline merge when the dataset P has size N, and by $F(N)$ the cost of our 4-d skyline algorithm. $G(N)$ equals $H(N)$ plus the overhead of sorting $SKY(P)$. Hence:

$$G(N) = O((N/B)\log_{M/B}(N/B)).$$

With the above, we solve the recurrence in Equation 3 as $F(N) = O((N/B)\log^2_{M/B}(N/B))$.

3.3 Higher dimensionalities

We are now ready to extend our technique to dimensionality $d \geq 5$, the core of which is to attack the following problem (that generalizes the skyline merges encountered in the preceding subsections). The input includes:

- A parameter h satisfying $0 \leq h \leq d-2$.
- A set of $s = \Theta(\min\{M^{1/(d-2)}, M/B\})$ slabs $\sigma_i(1), ..., \sigma_i(s)$ for each dimension $i \in [1, h]$ (there are h sets in total), if $h > 0$. No slab needs to be given for $h = 0$. Each set of $\sigma_i(1), ..., \sigma_i(s)$ is obtained by cutting the data space $\mathbb{R}^d$ with $s-1$ hyper-planes perpendicular to dimension i. We follow the convention that all points in $\sigma_i(j_1)$ have smaller coordinates on dimension i than those in $\sigma_i(j_2)$ for any $1 \leq j_1 < j_2 \leq s$.
- A set Π of points which are sorted in ascending order of their coordinates on dimension $h+1$. These points have the property that, for any $p_1, p_2 \in \Pi$, they do not dominate each other if they are covered by the same slab, namely, both p_1 and p_2 fall in a $\sigma_i(j)$ for some $i \in [1, h]$ and $j \in [1, s]$.

The objective is to output $SKY(\Pi)$ in ascending order of their coordinates on dimension $h+1$. We refer to the problem as *(h, d)-merge*.

Let $K = |\Pi|$. Our earlier analysis essentially has shown that $(1,3)$- and $(2,4)$-merges can both be settled in linear I/Os, while $(1,4)$-merge in $O((K/B)\log_{M/B}(K/B))$ I/Os. Next, we will establish a general result:

LEMMA 3.2. *The (h, d)-merge problem can be solved in $O((K/B)\log^{d-h-2}_{M/B}(K/B))$ I/Os.*

The subsequent discussion proves the lemma by handling $h = d-2$ and $h < d-2$ separately.

$h = d-2$. Our algorithm in this case performs a single scan of Π. At any time, we maintain s^h memory-resident values $\lambda(i_1, ..., i_h)$, where $1 \leq i_j \leq s$ for each $j \in [1, h]$. Specifically, $\lambda(i_1, ..., i_h)$ equals the lowest coordinate on dimension d of all the points already scanned that fall in $\sigma_1(i_1) \cap ... \cap \sigma_h(i_h)$; or $\lambda(i_1, ..., i_h) = \infty$ if no such point has been encountered yet. By remembering in memory the description of all slabs, we can update the corresponding $\lambda(i_1, ..., i_h)$ (if necessary) right after a point is read, without any extra I/O.

$SKY(\Pi)$ is empty at the beginning of the algorithm. Let p be the next point of Π found by the scan. Assume, without loss of generality, that p falls in $\sigma_1(i_1) \cap ... \cap \sigma_h(i_h)$ for some $i_1, ..., i_h$. We add p to $SKY(\Pi)$ if

$$p[d] < \lambda(j_1, ..., j_h), \forall j_1 < i_1, ..., j_h < i_h \quad (5)$$

where $p[d]$ is the d-th coordinate of p. With the experience from Equations 2 and 4, it is not hard to see that $p \in SKY(\Pi)$ if and only if Inequality 5 holds. Obviously, $SKY(\Pi)$ can be easily output in ascending order of the coordinates of dimension $h+1$ as this is the order by which points enter $SKY(\Pi)$.

The above process requires $O(s^h)$ memory to store the slab description and all the $\lambda(i_1, ..., i_h)$, plus an extra block as the input buffer of Π and output buffer of $SKY(\Pi)$, respectively. This is not a problem because $s^h = O(M)$. The algorithm terminates in $O(K/B)$ I/Os.

$h < d-2$. We deal with this scenario by converting the problem to $(h+1, d)$-merge, using a divide-and-conquer approach similar to transforming $(1,4)$-merge into $(2,4)$-merge in Section 3.2. Our approach is to generalize the algorithm `preMerge-4d` in Section 3.2. The resulting algorithm, named `preMerge`, solves the same problem as (h, d)-merge except that it outputs the points of $SKY(\Pi)$ in ascending order of their coordinates on dimension $h+2$. This is minor because $SKY(\Pi)$ can then be sorted again in $O((K/B)\log_{M/B}(K/B))$ I/Os to meet the order requirement of (h, d)-merge.

If $K \leq M$, `preMerge` trivially computes $SKY(\Pi)$ in memory. Otherwise, in linear I/Os the algorithm divides Π into $\Pi(1), ..., \Pi(s)$ of roughly the same size, such that points of $\Pi(i)$ have lower coordinates on dimension $h+1$ than those of $\Pi(j)$, for any $1 \leq i < j \leq s$. Naturally, each $\Pi(i)$ corresponds to a slab $\sigma_{h+1}(i)$, such that $\sigma_{h+1}(1), ..., \sigma_{h+1}(s)$ are separated by $s-1$ hyper-planes perpendicular to dimension $h+1$.

We then invoke `preMerge` recursively on $\Pi(i)$, feeding the same h sets of slabs $\{\sigma_i(1), ..., \sigma_i(s)\}$ of all $i \in [1, h]$. On return, we have obtained $\Sigma(i) = SKY(\Pi(i))$, with the points therein sorted ascendingly on dimension $h+2$. In $O(K/B)$ I/Os, $\Sigma(1), ..., \Sigma(s)$ can be merged into a single list Σ' where all points remain in ascending order of dimension $h+2$. At this point, we are facing a $(h+1, d)$-merge problem on Σ' and

```
merge(h, Π)
/* perform a (h, d)-merge on Π */
1. if h = d − 2 then
2.     compute SKY(Π) in O(|Π|/B) I/Os
       /* SKY(Π) now sorted by dimension h + 1 */
3. else
4.     SKY(Π) ← preMerge(h, Π)
5.     sort the points of SKY(Π) by dimension h + 1
6. return SKY(Π)
```

```
preMerge(h, Π)
1. if Π fits in memory then
2.     compute SKY(Π) in O(|Π|/B) I/Os
       /* sort the points of SKY(Π) in memory by
       dimension h + 2 */
3. else
4.     divide Π into Π(1), ..., Π(s) on dimension h + 1
5.     for i ← 1 to s do
6.         Σ(i) ← preMerge(h, Π(i))
7.     merge Σ(1), ..., Σ(s) into Σ′
       /* Σ′ now sorted by dimension h + 2 */
8.     SKY(Π) ← merge(h + 1, Σ′)
       /* SKY(Π) now sorted by dimension h + 2 */
9. return SKY(Π)
```

Figure 4: High-level description of the algorithms for performing a (h, d)-merge

$h+1$ sets of slabs $\{\sigma_i(1), ..., \sigma_i(s)\}$ of all $i \in [1, h+1]$. The converted problem is solved recursively as described above.

Running time of (h, d)-merge. The pseudocode of Figure 4 summarizes the main ideas of (h, d)-merge and `preMerge`. Let $H(h, K)$ be the cost of `preMerge`(h, Π) on a dataset Π of size K, and $G(h, K)$ the cost of a (h, d)-merge on Π. From the earlier discussion, we have:

$$G(h,K) = \begin{cases} O(K/B) & \text{if } h = d-2 \\ H(h,K) + O\left(\left(\frac{K}{B}\right)\log_{M/B}\left(\frac{K}{B}\right)\right) & \text{otherwise} \end{cases} \quad (6)$$

and, for $h \leq d-3$:

$$H(h,K) = \begin{cases} O(K/B) & \text{if } K < M \\ s \cdot H\left(h, \frac{K}{s}\right) + O\left(\frac{K}{B}\right) + G(h+1,K) & \text{otherwise} \end{cases} \quad (7)$$

where $s = \Omega((M/B)^{1/(d-2)})$. To solve the above recurrences, first notice that

$$H(d-3,K) = O((K/B)\log_{M/B}(K/B)) \quad (8)$$

which, together with Equation 6, implies that, for $h \leq d-3$:

$$G(h,K) = O(H(h,K)) \quad (9)$$

Hence, for $h \leq d-4$:

$$H(h,K) = \begin{cases} O(K/B) & \text{if } K < M \\ s \cdot H\left(h, \frac{K}{s}\right) + O(H(h+1,K)) & \text{otherwise} \end{cases}$$

From the above and Equation 8, we obtain $H(h, K) = O((K/B)\log^{d-h-2}_{M/B}(K/B))$ for all $h \leq d-3$. This, together with Equation 9 and the first case of Equation 6, completes the proof of Lemma 3.2.

Computing the skyline. Let P be a dataset that has been sorted in ascending order along dimension 1. We find $SKY(P)$ by simply performing a $(0, d)$-merge on P. By Lemma 3.2, the overall I/O complexity is $O((N/B)\log^{d-2}_{M/B}(N/B))$, which concludes the proof of Theorem 1.1.

Eliminating the general-position assumption. As mentioned by the remark in Section 1.3, the skyline problem is still well defined on a set P of points that are not in general position, namely, two points may have identical coordinates on some (but not all) dimensions. Next, we explain how to extend our algorithm to support such P.

The only part that needs to be clarified is how to deal with ties in sorting. Recall that, in several places of our algorithm, we need to sort a set $\Pi \subseteq P$ of points in ascending order of their coordinates on dimension i, where $1 \leq i \leq d$. The goal of tie-breaking is to ensure that if a point p_1 ranks *after* another point p_2, then p_1 cannot dominate p_2. This purpose can be achieved as follows. If p_1 and p_2 have the same i-th coordinate, we rank them lexicographically. Specifically, we first find the smallest j such that p_1 and p_2 have different coordinates on dimension j. Note that j must exist because P is a set, and hence, does not have duplicate points. If the j-th coordinate of p_1 is smaller than that of p_2, we rank p_1 earlier; otherwise, p_2 is ranked earlier.

4. CONCLUDING REMARKS

We have shown that, in the EM model, the skyline problem of any fixed dimensionality $d \geq 3$ can be settled in $O((N/B)\log^{d-2}_{M/B}(N/B))$ I/Os, where N is the dataset size, B is the size of a disk block, and M is the capacity of main memory.

Chan et al. [8] proposed the concept of *k-dominant skyline*, where k is a positive integer at most d. Intuitively, the dominance relation accompanying the new concept requires a point p_1 to be better than another p_2 only on k dimensions, instead of all dimensions. Specifically, p_1 is said to *k-dominate* p_2, denoted as $p_1 \prec_k p_2$, if:

$$\exists \text{ at least } k \text{ dimensions } i_1, ..., i_k \text{ s.t. } p_1[i_j] < p_2[i_j] \text{ for each } j = 1, ..., k.$$

The *k-dominant skyline* of P is the set of points in P that are not k-dominated by any other point in P. This problem can

be trivially solved by *BNL* in $O(N^2/(MB))$ I/Os. The existing k-dominant-skyline algorithms [8, 17, 25] are heuristic, and have the same complexity as *BNL* in the worst case.

For a fixed d, our technique can be utilized to settle this problem in $O((N/B)\log^{k-2}_{M/B}(N/B))$ I/Os for any $k \geq 3$. Let a *k-subspace* be the space $\mathbb{S}$ defined by k dimensions of $\mathbb{R}^d$. We say that a point $p \in P$ is in the skyline under $\mathbb{S}$, if p is a skyline point of the k-dimensional dataset obtained by projecting P onto $\mathbb{S}$. Clearly, there are $\binom{d}{k} = O(1)$ k-subspaces. It is easy to verify that p belongs to k-dominant skyline of P if and only if p is in the skyline under *all* k-subspaces. The proposed skyline algorithm allows us to find the skyline under each of the k-subspaces in totally $O((N/B)\log^{k-2}_{M/B}(N/B))$ I/Os, after which it is easy to extract the k-dominant skyline in $O((N/B)\log_{M/B}(N/B))$ I/Os. Finally, note that the 2-dominant skyline can be found in $O((N/B)\log_{M/B}(N/B))$ I/Os using similar ideas, whereas the 1-dominant skyline can be retrieved in $O(N/B)$ I/Os by scanning the dataset once (to get the minimum coordinate of the points in P on every dimension).

5. ACKNOWLEDGEMENTS

This work was supported by grants GRF 4173/08, GRF 4169/09, and GRF 4166/10 from HKRGC.

6. REFERENCES

[1] P. Afshani, J. Barbay, and T. M. Chan. Instance-optimal geometric algorithms. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 129–138, 2009.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[3] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Algorithms and Data Structures Workshop (WADS)*, pages 83–94, 1993.

[4] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM (CACM)*, 23(4):214–229, 1980.

[5] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. *Algorithmica*, 9(2):168–183, 1993.

[6] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. *Journal of the ACM (JACM)*, 25(4):536–543, 1978.

[7] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 421–430, 2001.

[8] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 503–514, 2006.

[9] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 717–816, 2003.

[10] H. K. Dai and X.-W. Zhang. Improved linear expected-time algorithms for computing maxima. In *Latin American Theoretical Informatics*, pages 181–192, 2004.

[11] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984.

[12] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *The VLDB Journal*, 16(1):5–28, 2007.

[13] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 714–723, 1993.

[14] R. Janardan. On the dynamic maintenance of maximal points in the plane. *Information Processing Letters (IPL)*, 40(2):59–64, 1991.

[15] S. Kapoor. Dynamic maintenance of maxima of 2-d point sets. *SIAM Journal of Computing*, 29(6):1858–1877, 2000.

[16] D. G. Kirkpatrick and R. Seidel. Output-size sensitive algorithms for finding maximal vectors. In *Symposium on Computational Geometry (SoCG)*, pages 89–96, 1985.

[17] M. Kontaki, A. N. Papadopoulos, and Y. Manolopoulos. Continuous k-dominant skyline computation on multidimensional data streams. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, pages 956–960, 2008.

[18] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of Very Large Data Bases (VLDB)*, pages 275–286, 2002.

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

[20] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 502–513, 2005.

[21] J. Matousek. Computing dominances in E^n. *Information Processing Letters (IPL)*, 38(5):277–278, 1991.

[22] M. D. Morse, J. M. Patel, and H. V. Jagadish.

Efficient skyline computation over low-cardinality domains. In *Proceedings of Very Large Data Bases (VLDB)*, pages 267–278, 2007.

[23] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems (TODS)*, 30(1):41–82, 2005.

[24] A. D. Sarma, A. Lall, D. Nanongkai, and J. Xu. Randomized multi-pass streaming skyline algorithms. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):85–96, 2009.

[25] M. A. Siddique and Y. Morimoto. K-dominant skyline computation by using sort-filtering method. In *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 839–848, 2009.

[26] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters (IPL)*, 6(3):80–82, 1977.

[27] J. S. Vitter. Algorithms and data structures for external memory. *Foundation and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.