Worst-case I/O-efficient Skyline Algorithms

CHENG SHENG
Chinese University of Hong Kong
and
YUFEI TAO
Korea Advanced Institute of Science and Technology

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 coordinate of the former is at most that of 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 point in P.

The main result of this article is that, for any fixed dimensionality $d \geq 3$, in external memory the skyline problem can be settled by performing $O((N/B)\log_{M/B}^{d-2}(N/B))$ I/Os in the worst case, where N is the cardinality of P, B the size of a disk block, and M the capacity of main memory. Similar bounds can also be achieved for computing several skyline variants, including the *k-dominant skyline*, *k-skyband*, and *α-skyline*. Furthermore, the performance can be improved if some dimensions of the data space have small domains. When the dimensionality d is not fixed, the challenge is to outperform the naive algorithm that simply checks all pairs of points in $P \times P$. We give an algorithm that terminates in $O((N/B)\log^{d-2} N)$ I/Os, thus beating the naive solution for any $d = O(\log N/\log\log N)$.

Categories and Subject Descriptors: F2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems

General Terms: Algorithms, theory

Additional Key Words and Phrases: 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 (by the definition of a set, no two points in P are identical). 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 the coordinate of p_1 is at most that of p_2 on

Author's address: C. Sheng (*csheng@cse.cuhk.edu.hk*), Department of Computer Science and Engineering, Chinese University of Hong Kong, Sha Tin, Hong Kong; Y. Tao (*taoyf@kaist.ac.kr*), Division of Web Science and Technology, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea.

This work was supported in part by (i) projects GRF 4169/09, 4166/10, 4165/11 from HKRGC, and (ii) the WCU (World Class University) program under the National Research Foundation of Korea, and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007).

Fig. 1. The skyline is $\{1, 5, 7\}$

every dimension, namely:

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

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 [Sarma et al. 2009]).

Ever since its debut in the database literature a decade ago [Borzsonyi et al. 2001], skyline computation has generated considerable interests in the database area (see [Sarma et al. 2009] 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* (i.e., 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 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 [Aggarwal and Vitter 1988], which has been successful in capturing the characteristics of algorithms dealing with massive data that do not fit in memory (see [Vitter 2006] 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 [Matousek 1991] 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 2d space, Kung et al. [Kung et al. 1975] 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 [Bentley 1980] developed an alternative algorithm achieving the same bounds as those of [Kung et al. 1975]. Kirkpatrick and Seidel [Kirkpatrick and Seidel 1985] presented algorithms whose running time is sensitive to the result size, and has the same complexity as the algorithms in [Bentley 1980; Kung et al. 1975] 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 [Bentley 1980; Kung et al. 1975] are already optimal in this class for $d = 2$ and 3 (see also some recent results on instance optimality due to Afshani et al. [Afshani et al. 2009]). For $d \geq 4$, Gabow et al. [Gabow et al. 1984] discovered an algorithm terminating in $O(N \log^{d-3} N \log\log N)$ time, whereas recently Chan et al. [Chan et al. 2011] improved the bound to $O(N \log^{d-3} N)$. Note, however, that the solutions of [Chan et al. 2011; Gabow et al. 1984] do not belong to the comparison class, due to their reliance on the features of the RAM model. Faster algorithms have been developed in some special circumstances where, for example, the data follow special distributions [Bentley et al. 1993; Bentley et al. 1978; Dai and Zhang 2004].

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 [Chan et al. 2011] in EM naively would entail $O(N \log^{d-3} N)$ I/Os, which amounts to reading the entire dataset $O(B \log^{d-3} N)$ times (B is at the order of thousands in practice). Hence, there is a genuine need to design I/O-oriented algorithms. For $d = 2$, using the findings of [Kung et al. 1975], 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 [Bentley 1980], 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.

The skyline of a dataset P can be trivially obtained by computing the cartesian

[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.

product $P \times P$ (i.e., by comparing all pairs of points in P). For any fixed d, the cartesian product can be produced by a *blocked nested loop (BNL)* in $\Theta(N^2/(MB))$ I/Os. It has been observed [Borzsonyi et al. 2001] 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. [Borzsonyi et al. 2001] 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. [Chomicki et al. 2003] works by sorting the points $p \in P$ in ascending order of $score(p)$, where *score* can be any function $\mathbb{R}^d \rightarrow \mathbb{R}$ 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. [Godfrey et al. 2007] 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 dimensions are independent, and the points of P distribute uniformly in the data space), provided that the memory size M is not too small. When the assumption does not hold, however, it remains unknown whether the cost of *LESS* is $o(N^2/(MB))$.

Sarma et al. [Sarma et al. 2009] 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,

method	I/O complexity	remark
[Kung et al. 1975]	$O(N \log^{d-2} N)$	
[Chan et al. 2011]	$O(N \log^{d-3})$	not in the comparison class.
[Bentley 1980]	$O((N/B) \log_2^{d-2}(N/M))$	adapted from Bentley's $O(N \log^{d-2} N)$ algorithm in RAM
BNL	$O(N^2/(MB))$	same for the *BNL* variant of [Borzsonyi et al. 2001]
DC [Borzsonyi et al. 2001]	$\Omega(N^2/(MB))$	
SFS [Chomicki et al. 2003]	$O(N^2/(MB))$	
LESS [Godfrey et al. 2007]	$O(N^2/(MB))$	
RAND [Sarma et al. 2009]	$O(\mu N/(MB))$ expected	μ is the number of points in the skyline, which can be $\Omega(N)$.
this article	$O((N/B) \log_{M/B}^{d-2}(N/B))$	

Table I. Comparison of skyline algorithms for fixed $d \geq 3$

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 pre-processing a dataset P into a structure that supports retrieval of the skyline without reading the whole dataset (see [Bartolini et al. 2008; Janardan 1991; Kapoor 2000; Kossmann et al. 2002; Lin et al. 2005; Papadias et al. 2005] and the references therein). In our context, these pre-computation-based methods do not have a notable advantage over the algorithms mentioned earlier.

1.3 Our results

Main result. We prove in this article:

THEOREM 1. *For any fixed $d \geq 3$, 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.*

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 2d space. Unlike the result of Godfrey et al. [Godfrey et al. 2007], 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. For any fixed d, Theorem 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 I 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 $s - 1$

[2]An algorithm paradigm proposed by Goodrich et al. [Goodrich et al. 1993] that can be regarded as the counterpart of plane-sweep in external memory.

hyper-planes 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. [Borzsonyi et al. 2001]. We overcome the obstacle by lowering the dimensionality to 3 gradually, and then settling the resulting 3d problem in linear I/Os. Our solution can also be regarded as the counterpart of Bentley's algorithm [Bentley 1980] in external memory.

We note that the 3d result of Theorem 1 was briefly mentioned to be achievable in [Goodrich et al. 1993], although no algorithmic details were given in that work. Our results for $d \geq 4$ appear in the literature for the first time to our knowledge.

Skyline variants. We show that performance bounds similar to that in Theorem 1 can also be achieved for computing several skyline variants, including *the k-dominant skyline* [Chan et al. 2006], *k-skyband* [Papadias et al. 2005], and *α-skyline* [Xia et al. 2008].

Low-cardinality domains. Let $[x]$ denote the set of integers $\{1, ..., x\}$. Our discussion has been assuming that each point is in the data space $\mathbb{R}^d$. Morse et al. [Morse et al. 2007] pointed out that, in practice, some dimensions may have small domains, e.g., the rating of a hotel may be an integer from 1 to 10. They raised the question whether the computation time can be reduced in such a scenario. Formally, let P be a set of N points in $[U]^t \times \mathbb{R}^{d-t}$ for some integer U, and some integer $t \in [0, d]$. In other words, the data space has t *short* dimensions with domain $[U]$, plus $d-t$ *long* dimensions with domain $\mathbb{R}$. The goal is still to compute $SKY(P)$ which is defined as in (1).

Assuming $t \geq d-1$ (i.e., at least $d-1$ dimensions are short), Morse et al. [Morse et al. 2007] solved the problem in $O(U^{d-1} + N/B)$ I/Os. In this article, we show:

THEOREM 2. *For any fixed $d \geq 3$, the skyline of N points in $[U]^t \times \mathbb{R}^{d-t}$ can be computed in*

—$O(\frac{N}{B} \log_{M/B} \frac{N}{B} + \frac{N}{B} \log^{d-2}_{M/B} \frac{U}{B})$ *I/Os if $t \geq d-1$;*

—$O(\frac{N}{B} \log^{d-3}_{M/B} \frac{U}{B} \cdot \log_{M/B} \frac{N}{B})$ *I/Os if $t = d-2$;*

—$O(\frac{N}{B} \log^{t}_{M/B} \frac{U}{B} \cdot \log^{d-t-2}_{M/B} \frac{N}{B})$ *I/Os if $t < d-2$.*

Again, the theorem focuses on $d \geq 3$ because the problem can be solved in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os in 2d space. Note that, for $t < d$, the above result is better than Theorem 1 when N is far greater than U. For small N, one can always revert to the algorithm of Theorem 1.

Non-fixed *d*. The skyline problem is challenging when the dimensionality d cannot be regarded as a constant, but instead, needs to play a part in the I/O complexity. In fact, d can even exceed M, i.e., the memory is not even large enough to hold all the coordinates of a single point, such that checking whether a point dominates another requires $O(d/B)$ I/Os. In this case, a tough competitor is the naive algorithm that simply checks all pairs of points and thus performs $O(dN^2/B)$ I/Os in total. Beating this naive solution becomes the primary goal. In this article, we

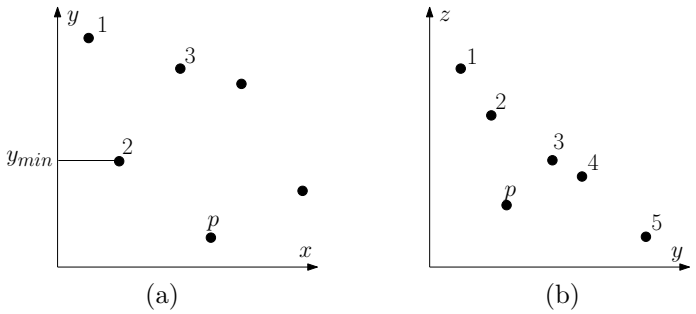

Fig. 2. Illustration of algorithms by Kung et al. (a) 2d, (b) 3d

establish:

THEOREM 3. *For any non-fixed $d \geq 3$, the skyline of N points in $\mathbb{R}^d$ can be computed in $O((N/B)\log^{d-2} N)$ I/Os.*

Note that the theorem improves the naive $O(dN^2/B)$ bound for all $d = O(\log N/\log\log N)$. We achieve the above result by observing that the analysis of [Kung et al. 1975] can be tightened. Specifically, it was shown there that, in internal memory, the skyline problem of a non-fixed d can be solved in $O(d^2 N \log^{d-2} N)$ time. By slightly modifying the algorithm of [Kung et al. 1975] and presenting a more careful analysis, we improve their bound to $O(N \log^{d-2} N)$. We then show that our analysis extends to external memory as well.

Remark. A short version of this article has appeared in [Sheng and Tao 2011]. That earlier work presented Theorem 1 and discussed k-dominant skylines, whereas all the other results mentioned above are additional materials of the current article.

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 2d solution of Kung et al. [Kung et al. 1975] can be easily adapted to work in the EM model. Second, our discussion of existing algorithms for $d \geq 3$ not only clarifies several 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 [Kung et al. 1975] and why a similar bound also holds in the I/O context.

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

2d. The skyline $SKY(P)$ of a set P of 2d 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.

3d. Suppose that we have sorted P in ascending order of their x-coordinates. Similar to the 2d 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 2d 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 2d 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 2d 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 [Kung et al. 1975] 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 to externalize 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 [Goodrich et al. 1993]) 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 [Bentley 1980] 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. [Kung et al. 1975] and Bentley [Bentley 1980] 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 hyper-plane 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.

Externalization of the algorithms of Kung et al. [Kung et al. 1975] and Bentley [Bentley 1980] 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, 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 [Bentley 1980] or [Kung et al. 1975] 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. [Kung et al. 1975] proved that any 2d 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. [Kung et al. 1975] 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 [Aggarwal and Vitter 1988] 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 2d skylines in external memory. Note that the same bound also holds in higher-dimensional space where the problem is no easier than in the 2d 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. [Arge et al. 1993].

3. OUR SKYLINE ALGORITHM

This section will present a new algorithm for solving the skyline problem when the dimensionality d is a fixed constant. Let us agree on some more terminologies.

When we say a set S of points is "sorted on a certain dimension", we mean that the points of S are sorted by their coordinates of that dimension. By "partitioning S into s sets $S(1), ..., S(s)$ along a dimension, say x," we mean cutting the data space into *slabs* $\sigma(1), ..., \sigma(s)$ using $s-1$ hyper-planes perpendicular to the x-dimension, such that $S(i) = S \cap \sigma(i)$ for each $i \in [1, s]$. Furthermore, we follow the convention that points in $S(i)$ have smaller x-coordinates than those in $S(j)$ for any $i < j$. Also, by "synchronously scan $S(1), ..., S(s)$ on, for example, the x-dimension", we mean fetching the points of $S(1) \cup ... \cup S(s)$ in ascending order of their x-coordinates.

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 3d 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 4d description of our algorithm can then be easily extended to general d, which is the topic of Section 3.3. For simplicity, in the above subsections, we will assume that P is in *general position*, i.e., no two points of P share the same coordinate on any dimension. This assumption will be removed in Section 3.4.

3.1 3d

Our algorithm accepts as input a dataset P sorted on the x-dimension. 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 $O(N/B)$.

If $N > M$, we divide P into $s = \Theta(M/B)$ partitions $P(1), ..., P(s)$ with roughly the same size[3]. 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 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

[3] "Roughly the same size" means that the size of each $P(i)$ is the same up to a constant factor.

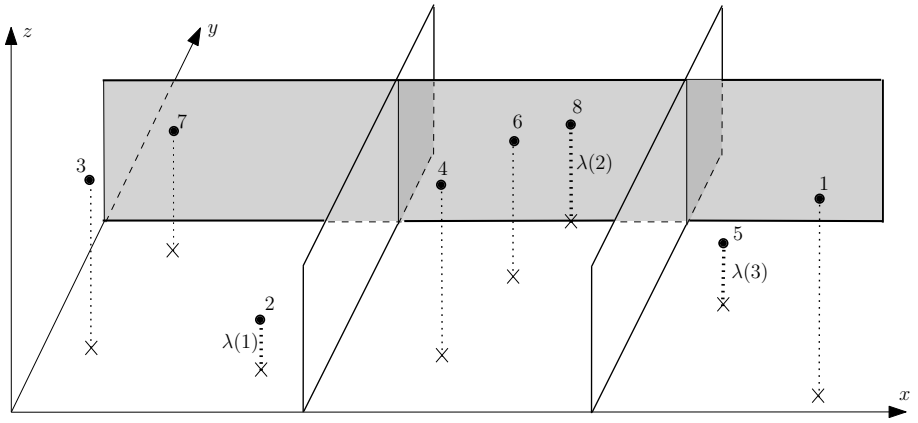

Fig. 3. Illustration of 3d skyline merge. The value of s is 3. Only the points already encountered are shown. Points are labeled in ascending order of their y-coordinates (which is also the order they are fetched). Point 8 is the last one seen. Each cross is the projection of a point in the x-y plane. $\Sigma(1)$ 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 (2) on $j = 1$).

lemma below shows the correctness of this rule.

LEMMA 1. *$p \in SKY(P)$ if and only if (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 (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

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

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 4d

To find the skyline of a 4d dataset P, we proceed as in the 3d 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 that P has been partitioned into $P(1), ..., P(s)$ with s *slabs* $\sigma_1(1), ..., \sigma_1(s)$ separated by $s-1$ hyper-planes orthogonal to dimension 1. 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 3d space with $O(N/B)$ I/Os where $N = |P|$. In 4d 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 $SKY(\Pi)$ (output by `preMerge-4d`) 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 $\Theta(M)$ 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, along dimension 2. 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 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 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 4d 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 (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 $0 \leq h \leq d-2$.

—(Applicable only to $h > 0$) 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). Each set of $\sigma_i(1), ..., \sigma_i(s)$ is obtained by cutting $\mathbb{R}^d$ 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 sorted 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*.

Set $K = |\Pi|$. Our earlier analysis 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 2. *For any fixed d, the (h, d)-merge problem on N d-dimensional points 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

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

where $p[d]$ is the d-th coordinate of p. With the experience from (2) and (4), it is not hard to see that $p \in SKY(\Pi)$ if and only if (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 along dimension $h+1$. 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 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 sorted on dimension $h + 2$. At this point, we are facing a $(h + 1, d)$-merge problem on Σ' and $h + 1$ 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 execution flows of (h, d)-merge (i.e., function `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:

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

```
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, ensuring that the points of
       SKY(Π) be sorted by dimension h + 2
4. else
5.     divide Π into Π(1), ..., Π(s) on dimension h + 1
6.     for i ← 1 to s do
7.         Σ(i) ← preMerge(h, Π(i))
8.     merge Σ(1), ..., Σ(s) into Σ′
       /* Σ′ now sorted by dimension h + 2 */
9.     SKY(Π) ← merge(h + 1, Σ′)
       /* SKY(Π) now sorted by dimension h + 2 */
10. return SKY(Π)
```

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

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 (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 (8), we obtain $H(h,K) = O((K/B)\log_{M/B}^{d-h-2}(K/B))$ for all $h \leq d-3$. This, together with (9) and the first case of (6), completes the proof of Lemma 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 2, the overall I/O complexity is $O((N/B)\log^{d-2}_{M/B}(N/B))$, which concludes the proof of Theorem 1.

3.4 Eliminating the general-position assumption

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. VARIANTS OF THE SKYLINE PROBLEM

This section will explain how to extend our algorithm of Section 3 to settle some variants of the skyline problem in a worst-case efficient manner.

***k*-dominant skyline.** Chan et al. [Chan et al. 2006] 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 [Chan et al. 2006] 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).

***k*-skyband.** Given a set P of d-dimensional points, a *k-skyband* consists of all the points $p \in P$ such that p is dominated by less than k points in P. Hence, the skyline of P is simply the 1-skyband, and in general, the k-skyband is a subset of the $(k+1)$-skyband. This notion, introduced by Papadias et al. [Papadias et al. 2005], provides a user with a richer set of choices (than the conventional skyline), and yet guarantees that any point offered is among the k elements in P that minimize at least one monotone preference function (i.e., a *top-k set*).

For any

$$k = O(M^{1-\epsilon})$$

where ϵ is any positive constant smaller than 1, our algorithm can be adapted to find the k-skyband of P in $O(\frac{N}{B} \log_{M/B}^{d-2} \frac{N}{B})$ I/Os, where $N = |P|$. Next, we explain this only for $d = 3$ because the generalization to higher d is straightforward.

Consider that P has already been sorted on the x-dimension. Set

$$s = \Theta(\min\{M^{\epsilon}, M/B\}).$$

As in our skyline algorithm, divide P into $P(1), ..., P(s)$ along the x-dimension of roughly the same size. Recurse on each $P(i)$ for $i \in [1, s]$ to find its k-skyband $\Sigma(i)$. On return, we require that $\Sigma(i)$ should have been sorted on the y-dimension. Furthermore, each point $p \in \Sigma(i)$ should be associated with a value $cnt(p)$ that equals the number of points in $P(i)$ dominating p. Clearly, $cnt(p) < k$.

Synchronously scan $\Sigma(1), ..., \Sigma(s)$ on the y-dimension. For each $\Sigma(i)$, record the k smallest z-coordinates of the points of $\Sigma(i)$ that have been seen. Denote those coordinates as $\lambda_1(i), ..., \lambda_k(i)$ in ascending order ($\lambda_j(i) = \infty$ if less than j points have been encountered in $\Sigma(i)$). Let $p \in \Sigma(i)$ for some i be the next point fetched by the synchronous scan. We find the number c of values smaller than $p[3]$ among $\lambda_{j'}(j)$ for all $1 \leq j' \leq k$ and $1 \leq j < i$, and increase $cnt(p)$ by c. If $cnt(p)$ is at least k now, p is discarded. The remaining points are written to the file sorted on the y-dimension.

The choice of s ensures that the memory, besides storing sk z-coordinates, can still accommodate a block as the input buffer for each $\Sigma(i)$, as well as a block as the output buffer.

Dominance filtering. Let us consider the following problem. Let Π and P be two sets of d-dimensional points. We want to report all the points in P that are not dominated by any point in Π. We call this problem *dominance filtering*. As an application in practice, imagine that Π (P) represents a set of hotels of a company C_1 (C_2). The output of dominance filtering can be understood as those hotels of C_2 that are not superseded by any hotel of C_1. Besides its own practical usefulness, the problem is also fundamental to solving some other problems, as will be clear later.

Set $N = |\Pi| + |P|$. Below we show how to slightly modify our skyline algorithm to solve the dominance filtering problem in $O(\frac{N}{B} \log_{M/B}^{d-2} \frac{N}{B})$ I/Os. We will focus on 3d because the same idea extends to higher d. Let $S = \Pi \cup P$, assuming for simplicity that no point of Π coincides with a point in P.

Consider that S has been sorted on the x-dimension. Similar to our skyline algorithm, partition S into $S(1), ..., S(s)$ of roughly the same size along the x-

dimension, where $s = \Theta(M/B)$. Let $\Pi(i) = \Pi \cap S(i)$ and $P(i) = P \cap S(i)$. Recurse on each $S(i)$ such that, on return, all those points in $P(i)$ that are dominated by some point in $\Pi(i)$ have been eliminated. Let $P'(i)$ be the set of points of $P(i)$ that remain. We require that, on return from the recursion, $S'(i) = \Pi(i) \cup P'(i)$ should be sorted on y-dimension. Synchronously scan $S'(1), ..., S'(s)$ on the y-dimension. At any moment, maintain $\lambda(i)$ for $i \in [1, s]$ as the smallest z-coordinate of the points in $\Pi(i)$ that have been encountered ($\lambda(i) = \infty$ if no such point has been seen). When a point $p \in P'(i)$ for some i has been fetched, discard it if its z-coordinate is larger than any of $\lambda(1), ..., \lambda(i-1)$.

α-skyline. Next, we will put the above dominance filtering algorithm to an immediate use (we will see another use later in Section 6). Given two points $p_1, p_2 \in \mathbb{R}^d$, we say that p_1 *α-dominates* p_2 if

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

where α is a real value. Intuitively, a positive α makes it easier for p_1 to dominate p_2, whereas a negative α makes it more difficult. In any case, note that it is possible for p_1, p_2 to mutually α-dominate each other. Given a set P of points in $\mathbb{R}^d$, its *α-skyline* is the set of points in P that are not α-dominated by any other point in P. This notion was proposed by Xia et al. [Xia et al. 2008] as a tool to adjust the number of points returned to the user.

Our dominance filtering algorithm can be used to find the α-skyline of P in $O(\frac{N}{B} \log_{M/B}^{d-2} \frac{N}{B})$ I/Os, where $N = |P|$. We only need to create a point set Π where each point is obtained from a distinct point $p \in P$ by reducing its coordinate by α on all dimensions. Then, we simply return those points in P that are not dominated by any point in Π (note that the *dominance* here is defined as in conventional skylines, instead of α-dominance).

5. LOW-CARDINALITY DOMAINS

We proceed to discuss the skyline problem that Theorem 2 concerns, where the data space has t *short* dimensions with domain $[U]$, plus $d - t$ *long* dimensions with domain $\mathbb{R}$. We focus on $t \geq 1$, and without loss of generality, assume that dimensions 1, ..., t are short, and dimensions $t+1, ..., d$ are long.

Set $s = \Theta(\min\{M^{1/(d-2)}, M/B\})$. Let us generalize the (h, d)-merge problem of Section 3.3 to the following *(h, d)-general-merge problem*. The input includes

—An integer $h \in [0, d-2]$.

—A set Π of points sorted on dimension $h+1$.

—(Applicable only to $h > 0$) for each $i \in [1, h]$, one of the following is true:
  —dimension i has a domain with size at most s. In this case we say that the dimension is *exhausted*, and let $v_i(1), ..., v_i(s)$ be the values in the domain in ascending order.
  —otherwise, we are given s slabs $\sigma_i(1), ..., \sigma_i(s)$ that are separated by $s-1$ hyperplanes perpendicular to dimension i, such that any two points in the same slab $\sigma_i(j)$ (for any $j \in [1, s]$) do not dominate each other.

The output is $SKY(\Pi)$ sorted on dimension $h+1$. With $K = |\Sigma|$, we will show:

LEMMA 3. *The (h, d)-general-merge problem can be solved in*

—$O\left(\frac{K}{B}\log_{M/B}^{t-1-h}\frac{U}{B}\cdot\log_{M/B}\frac{K}{B}+\frac{K}{B}\log_{M/B}^{t-h}\frac{U}{B}\cdot\log_{M/B}^{d-t-2}\frac{K}{B}\right)$ *I/Os if* $t\leq d-2$;

—$O\left(\frac{K}{B}\log_{M/B}^{d-2-h}\frac{U}{B}\right)$ *I/Os if* $t\geq d-1$.

Note that Theorem 2 follows directly from the above lemma by setting h to 0, and including the cost of sorting the dataset on dimension 1. We first explain how to perform an (h, d)-general-merge, before proving Lemma 3.

***h* = *d* − 2.** Given $i_1, ..., i_h$ each of which ranges from 1 to s, define $cell(i_1, ..., i_h)$ as the region $\gamma_1(i_1)\cap ...\cap\gamma_h(i_h)$, where $\gamma_j = v_j$ if dimension j is exhausted, or $\gamma_j = \sigma_j$ otherwise. Scan the points of Π in ascending order of their coordinates on dimension $d-1$. Meanwhile, maintain $\lambda(i_1, ..., i_h)$ as the smallest coordinate on dimension d of all the points in $cell(i_1, ..., i_h)$ that have been seen. The rest of the algorithm is the same as the one in Section 3.3 for the case $h = d-2$.

***h* < *d* − 2.** We will describe an algorithm named `preMerge-gen` that solves the same problem as (h, d)-general-merge, except that on return $SKY(\Pi)$ should have been sorted on dimension $h+2$. To settle (h, d)-general-merge, we first apply `preMerge-gen` and then sort $SKY(\Pi)$ on dimension $h+1$. Algorithm `preMerge-gen` works as follows:

—If dimension $h+1$ is not exhausted, `preMerge-gen` behaves like `preMerge`:
  —If dimension $h+1$ is long, it divides Π along this dimension into $\Pi(1), ..., \Pi(s)$ of roughly the same size, which correspond to slabs $\sigma_{h+1}(1), ..., \sigma_{h+1}(s)$ respectively, as are separated by $s-1$ hyper-planes perpendicular to dimension $h+1$.
  —If dimension $h+1$ is short, let D be the size of its domain. `preMerge-gen` partitions the data space into slabs $\sigma_{h+1}(1), ..., \sigma_{h+1}(s)$ with $s-1$ hyper-planes perpendicular to dimension $h+1$ (suppose, as before, that the points in $\sigma_{h+1}(i)$ have smaller coordinates on dimension $h+1$ than those in $\sigma_{h+1}(j)$ for any $i < j$). Each of $\sigma_{h+1}(1), ..., \sigma_{h+1}(s)$ contains $\Theta(D/s)$ values of dimension $h+1$. Accordingly, Π is divided into $\Pi(1), ..., \Pi(s)$ such that $\Pi(i) = \Pi\cap\sigma_{h+1}(i)$ for each $i\in[1, s]$.

  In either case, we recursively invoke `preMerge-gen` on each $\Pi(i)$, treating $\sigma_{h+1}(i)$ as the data space. After the recursion, we finish with an $(h+1, d)$-general-merge as in handling $h < d-2$ in Section 3.3.

—If dimension $h+1$ is exhausted, we simply sort Π on dimension $h+2$, and convert the problem to $(h+1, d)$-general-merge (by feeding the domain values of dimension $h+1$ directly).

Analysis. We will prove Lemma 3 first for $t\leq d-2$. Denote by $G_{short}(h, D, K)$ the cost of an (h, d)-general-merge on Π when $h < t$ (i.e., dimension $h+1$ is short), where D is the domain size of dimension $h+1$. Hence, $D\leq U$. If $h < d-2$, define $H_{short}(h, D, K)$ to be the cost of the corresponding `preMerge-gen`. Similarly, denote by $G(h, K)$ the cost of (h, d)-general-merge when $h\geq t$.

The analysis of $G(h, K)$ is exactly the same as in Section 3.3, namely:

$$G(h, K) = O\left(\frac{K}{B} \log_{M/B}^{d-h-2} \frac{K}{B}\right). \tag{10}$$

It follows from the description of our algorithm that for all $h < t$:

$$G_{short}(h, D, K) = H_{short}(h, D, K) + O\left(\frac{K}{B} \log_{M/B} \frac{D}{B}\right) \tag{11}$$

Note that the second term on the right hand side of (11) corresponds to the cost of sorting the $SKY(\Pi)$ output by `preMerge-gen` on a dimension with domain size D. The cost of this is $O((K/B) \log_{M/B}(D/B))$ I/Os using the algorithm in [Arge et al. 1993], as opposed to $O((K/B) \log_{M/B}(K/B))$. Furthermore:

$$\begin{aligned} & H_{short}(t-1, D, K) \\ &= \begin{cases} O\left(\frac{K}{B} \log_{M/B} \frac{K}{B}\right) + G(t, K) & \text{if } D \le s \\ \sum_{i=1}^{s} H_{short}\left(t-1, \frac{D}{s}, |\Pi(i)|\right) + G(t, K) & \text{otherwise} \end{cases} \\ \text{(by (10))} &= \begin{cases} O\left(\frac{K}{B} \log_{M/B} \frac{K}{B}\right) + O\left(\frac{K}{B} \log_{M/B}^{d-t-2} \frac{K}{B}\right) & \text{if } D \le s \\ \sum_{i=1}^{s} H_{short}\left(t-1, \frac{D}{s}, |\Pi(i)|\right) + O\left(\frac{K}{B} \log_{M/B}^{d-t-2} \frac{K}{B}\right) & \text{otherwise} \end{cases} \end{aligned} \tag{12}$$

and for $h \le t-2$:

$$\begin{aligned} & H_{short}(h, D, K) \\ &= \begin{cases} O\left(\frac{K}{B} \log_{M/B} \frac{D}{B}\right) + G_{short}(h+1, U, K) & \text{if } D \le s \\ \sum_{i=1}^{s} H_{short}\left(h, \frac{D}{s}, |\Pi(i)|\right) + G_{short}(h+1, U, K) & \text{otherwise} \end{cases} \end{aligned} \tag{13}$$

Solving Equation (12) gives:

$$H_{short}(t-1, U, K) = O\left(\frac{K}{B} \log_{M/B} \frac{K}{B} + \frac{K}{B} \log_{M/B} \frac{U}{B} \cdot \log_{M/B}^{d-t-2} \frac{K}{B}\right). \tag{14}$$

From (11), (13) and (14), we can derive that, for $h \le t-2$:

$$\begin{aligned} & H_{short}(h, U, K) \\ &= O\left(\frac{K}{B} \log_{M/B}^{t-1-h} \frac{U}{B} \cdot \log_{M/B} \frac{K}{B} + \frac{K}{B} \log_{M/B}^{t-h} \frac{U}{B} \cdot \log_{M/B}^{d-t-2} \frac{K}{B}\right) \end{aligned}$$

The proof of Lemma 3 for $t \ge d-1$ is much simpler. First, we no longer need $G(h, K)$, and $G_{short}(d-2, D, K)$ equals simply $O(K/B)$. This, together with (11) and (13), leads to

$$H_{short}(h, U, K) = O\left(\frac{K}{B} \log_{M/B}^{d-2-h} \frac{U}{B}\right).$$

6. NON-FIXED DIMENSIONALITY

Next, we present our results for skyline computation in $\mathbb{R}^d$ where d is not a fixed constant, but a parameter in asymptotic complexities. Section 6.1 first shows how to improve the classic result of [Kung et al. 1975] by a factor of $O(d^2)$ in internal memory, and then, Section 6.2 extends the analysis to external memory.

6.1 An improved algorithm in internal memory

We first slightly modify the algorithm of [Kung et al. 1975] for solving the dominance filtering problem in memory. As defined in Section 4, in this problem, we are given a pair (Π, P), where Π (P) is a set of d-dimensional points. The goal is to report those points of P that are not dominated by any point in Π. Let $S = \Pi \cup P$, and refer to $N = |S|$ as the *problem size*. We consider that S has been sorted on dimension 1.

If $d \leq 5$, we settle the problem in $O(N \log^{d-2} N)$ time using the algorithm of [Kung et al. 1975]. Next, consider $d \geq 6$. If N is below a constant, the problem can be trivially solved in $O(d)$ time. Otherwise, we partition S into S_1 and S_2 of roughly the same size along dimension 1. This can be done in $O(dN)$ time using the algorithm of [Blum et al. 1973]. Let $\Pi_1 = \Pi \cap S_1, \Pi_2 = \Pi \cap S_2, P_1 = P \cap S_1$, and $P_2 = P \cap S_2$. We then recursively solve the dominance filtering problem on (Π_1, P_1) and (Π_2, P_2), respectively. The output of (Π_1, P_1) can be directly returned. Let P_2' be the output of (Π_2, P_2). We sort $\Pi_1 \cup P_2'$ on dimension 2, recursively solve the dominance filtering problem on (Π_1, P_2') in the $(d-1)$-dimensional subspace excluding dimension 1, and return the output.

It was shown in [Kung et al. 1975] that the dominance problem can be settled in $O(d^2 N \log^{d-2} N)$ time. Below we prove a better result:

LEMMA 4. *For any non-fixed d, our dominance filtering algorithm finishes in $O(N \log^{d-2} N)$ time.*

PROOF. All the logarithms in this proof have base 2. Let $T_d(N)$ be the time of solving the d-dimensional dominance problem when the problem size is N. By the result of [Kung et al. 1975], the lemma is correct when $d \leq 5$. Next, we will prove by induction that, for any $d \geq 6$ and N greater than a constant, there exists a constant $c > 0$ such that

$$T_d(N) \leq cN \log^{d-2} N + cdN. \tag{15}$$

Note that the inequality obviously holds if N is at most a constant, regardless of d. Assuming that (15) is true when either $N < N' \wedge d \leq d'$ or $N \leq N' \wedge d < d'$ holds, we will show that it is also true for $N = N'$ and $d = d'$, where $N' \geq 16$ and $d' \geq 6$ are two integers. Once this is done, Lemma 4 follows from the fact that $N \log^{d-2} N + dN \leq 2N \log^{d-2} N$ for $d \geq 4$ and $N \geq 4$.

Our algorithm ensures that, with some constant $c' > 0$:

$$\begin{aligned} T_{d'}(N') &\leq 2 \cdot T_{d'}(N'/2) + T_{d'-1}(N') + c'd'N' \log(d'N') \\ \text{(by inductive assumption)} &\leq cN' \log^{d'-2}(N'/2) + cd'N' + cN' \log^{d'-3} N' \\ &\quad + c(d'-1)N' + c'd'N' \log N' + c'd'N' \log d' \end{aligned}$$

Our goal is to show that the right hand side of the above is at most the right hand side of (15) when $N = N'$ and $d = d'$, namely:

$$\begin{aligned} &(\log N' - 1)^{d'-2} + \log^{d'-3} N' + (d'-1) + \left(\frac{c'}{c}\right) d' \log N' + \left(\frac{c'}{c}\right) d' \log d' \\ &\leq \log^{d'-2} N' \end{aligned}$$

Introducing $x = \log N' - 1$, we rewrite the above into

$$x^{d'-2} + (x+1)^{d'-3} + (d'-1) + \left(\frac{c'}{c}\right) d'(x+1) + \left(\frac{c'}{c}\right) d' \log d' \leq (x+1)^{d'-2} \Leftrightarrow$$

$$x^{d'-2} + \sum_{i=0}^{d'-3} \binom{d'-3}{i} x^i + (d'-1) + \left(\frac{c'}{c}\right) d'(x+1) + \left(\frac{c'}{c}\right) d' \log d' \leq \sum_{i=0}^{d'-2} \binom{d'-2}{i} x^i \Leftrightarrow$$

$$(d'-1) + \left(\frac{c'}{c}\right) d'(x+1) + \left(\frac{c'}{c}\right) d' \log d' \leq \sum_{i=0}^{d'-3} \left(\binom{d'-2}{i} - \binom{d'-3}{i} \right) x^i$$

Next, setting $c = c'$, we will complete the proof by showing that the above holds for $d' \geq 6$ and $x = \log N' - 1 \geq 3$. In fact, we will prove a stronger argument that

$$\begin{aligned} &(d'-1) + \left(\frac{c'}{c}\right) d'(x+1) + \left(\frac{c'}{c}\right) d' \log d' \\ &\leq \left(\binom{d'-2}{2} - \binom{d'-3}{2} \right) x^2 + \left(\binom{d'-2}{d'-3} - \binom{d'-3}{d'-3} \right) x^{d'-3} \end{aligned}$$

First:

$$\left(\binom{d'-2}{2} - \binom{d'-3}{2} \right) x^2 = (d'-3)x^2$$

which is at least $d'(x+1)$ for any $d' \geq 6$ (recall that $x \geq 3$). Second:

$$\left(\binom{d'-2}{d'-3} - \binom{d'-3}{d'-3} \right) x^{d'-3} \geq (d'-3)2^{d'-3}$$

which is at least $d' - 1 + d' \log d'$ for any $d' \geq 6$. □

Equipped with our dominance filtering algorithm, we find the skyline of a set P of points as follows. First, divide P into P_1 and P_2 of roughly the same size along the first dimension, and compute $SKY(P_1)$ and $SKY(P_2)$ by recursion. Then, we return $SKY(P_1)$ directly, and the output of the dominance filtering on $(SKY(P_1), SKY(P_2))$. Thus, we arrive at:

THEOREM 4. *For any non-fixed d, the skyline of N d-dimensional points can be computed in time $O(N \log^{d-2} N)$.*

Note that the above result improves the naive algorithm that checks the dominance of each pair of points in P (and hence incurs $O(dN^2)$ time) for all $d = O(\log N / \log \log N)$.

6.2 Externalizing the algorithm

Our dominance filtering algorithm can be easily adapted to work in external memory. Specifically, if $d \leq 5$, we solve the problem using our solution in Section 4 in $O(\frac{N}{B} \log_{M/B}^{d-2} \frac{N}{B})$ I/Os. The subsequent discussion concentrates on $d \geq 6$. If N is at most a constant, we settle the problem in $O(d/B)$ I/Os by simply checking each pair of $\Pi \times P$. Otherwise, we proceed in exactly the same way as described in the previous subsection. Two details are worth mentioning. First, partitioning S into S_1 and S_2 can be done in $O(dN/B)$ I/Os using the algorithm of [Aggarwal and

Vitter 1988]. Second, sorting $\Pi_1 \cup P_2'$ requires $O((dN/B)\log_{M/B}(dN/B))$ I/Os. Adapting the proof of Lemma 4 in a straightforward manner shows that our algorithm terminates in $O((N/B)\log^{d-2} N)$ I/Os. Theorem 3 can then be obtained in the same manner as deriving Theorem 4 from Lemma 4.

7. CONCLUSIONS

This article presents external memory algorithms for solving the skyline problem and its variants in a worst-case efficient manner. For any fixed $d \geq 3$, we solved the skyline problem in $O((N/B)\log^{d-2}_{M/B}(N/B))$ I/Os if the input contains N points from $\mathbb{R}^d$. Analagous results were also established for computing several variants of the skyline, including the k-dominant skyline, the k-skyband, and the α-skyline. We also showed that the running time can be improved if some dimensions have small domains.

When d is not fixed, beating the naive $O(dN^2/B)$ bound becomes the main challenge. We partially solved the task with an algorithm whose running time is $o(dN^2/B)$ when $d = O(\log N/\log\log N)$. It still remains open how to achieve this purpose for larger d.

Another interesting direction for future work is to study structures that can be deployed to compute a skyline without reading the underlying dataset in its entirety. Such a structure must also support point insertions and/or deletions. The 2d version of the problem has been well studied in internal memory (see a list of results in [Brodal and Tsakalidis 2011]), while it still remains elusive in external memory.

REFERENCES

Afshani, P., Barbay, J., and Chan, T. M. 2009. Instance-optimal geometric algorithms. In *FOCS*. 129–138.

Aggarwal, A. and Vitter, J. S. 1988. The input/output complexity of sorting and related problems. *CACM 31,* 9, 1116–1127.

Arge, L., Knudsen, M., and Larsen, K. 1993. A general lower bound on the I/O-complexity of comparison-based algorithms. In *WADS*. 83–94.

Bartolini, I., Ciaccia, P., and Patella, M. 2008. Efficient sort-based skyline evaluation. *TODS 33,* 4.

Bentley, J. L. 1980. Multidimensional divide-and-conquer. *CACM 23,* 4, 214–229.

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

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

Blum, M., Floyd, R. W., Pratt, V. R., Rivest, R. L., and Tarjan, R. E. 1973. Time bounds for selection. *JCSS 7,* 4, 4 48–461.

Borzsonyi, S., Kossmann, D., and Stocker, K. 2001. The skyline operator. In *ICDE*. 421–430.

Brodal, G. S. and Tsakalidis, K. 2011. Dynamic planar range maxima queries. In *ICALP*. 256–267.

Chan, C. Y., Jagadish, H. V., Tan, K.-L., Tung, A. K. H., and Zhang, Z. 2006. Finding k-dominant skylines in high dimensional space. In *SIGMOD*. 503–514.

Chan, T. M., Larsen, K. G., and Patrascu, M. 2011. Orthogonal range searching on the ram, revisited. In *SoCG*. 1–10.

Chomicki, J., Godfrey, P., Gryz, J., and Liang, D. 2003. Skyline with presorting. In *ICDE*. 717–816.

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

Gabow, H. N., Bentley, J. L., and Tarjan, R. E. 1984. Scaling and related techniques for geometry problems. In *STOC*. 135–143.

Godfrey, P., Shipley, R., and Gryz, J. 2007. Algorithms and analyses for maximal vector computation. *VLDB J. 16,* 1, 5–28.

Goodrich, M. T., Tsay, J.-J., Vengroff, D. E., and Vitter, J. S. 1993. External-memory computational geometry. In *FOCS*. 714–723.

Janardan, R. 1991. On the dynamic maintenance of maximal points in the plane. *IPL 40,* 2, 59–64.

Kapoor, S. 2000. Dynamic maintenance of maxima of 2-d point sets. *SIAM J. of Comp. 29,* 6, 1858–1877.

Kirkpatrick, D. G. and Seidel, R. 1985. Output-size sensitive algorithms for finding maximal vectors. In *SoCG*. 89–96.

Kossmann, D., Ramsak, F., and Rost, S. 2002. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*. 275–286.

Kung, H. T., Luccio, F., and Preparata, F. P. 1975. On finding the maxima of a set of vectors. *JACM 22,* 4, 469–476.

Lin, X., Yuan, Y., Wang, W., and Lu, H. 2005. Stabbing the sky: Efficient skyline computation over sliding windows. In *ICDE*. 502–513.

Matousek, J. 1991. Computing dominances in E^n. *IPL 38,* 5, 277–278.

Morse, M. D., Patel, J. M., and Jagadish, H. V. 2007. Efficient skyline computation over low-cardinality domains. In *VLDB*. 267–278.

Papadias, D., Tao, Y., Fu, G., and Seeger, B. 2005. Progressive skyline computation in database systems. *TODS 30,* 1, 41–82.

Sarma, A. D., Lall, A., Nanongkai, D., and Xu, J. 2009. Randomized multi-pass streaming skyline algorithms. *PVLDB 2,* 1, 85–96.

Sheng, C. and Tao, Y. 2011. Finding skylines in external memory. In *PODS*. 107–116.

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

Xia, T., Zhang, D., and Tao, Y. 2008. On skylining with flexible dominance relation. In *ICDE*. 1397–1399.