Maximal Vector Computation in Large Data Sets

Parke Godfrey[1] Ryan Shipley[2] Jarek Gryz[1]

[1]York University
Toronto, ON M3J 1P3, Canada
{godfrey, jarek}@cs.yorku.ca

[2]The College of William and Mary
Williamsburg, VA 23187-8795, USA

Abstract

Finding the maximals in a collection of vectors is relevant to many applications. The maximal set is related to the convex hull—and hence, linear optimization—and nearest neighbors. The maximal vector problem has resurfaced with the advent of skyline queries for relational databases and skyline algorithms that are external and relationally well behaved.

The initial algorithms proposed for maximals are based on divide-and-conquer. These established good average and worst case asymptotic running times, showing it to be $\mathcal{O}(n)$ average-case, where n is the number of vectors. However, they are not amenable to externalizing. We prove, furthermore, that their performance is quite bad with respect to the dimensionality, k, of the problem. We demonstrate that the more recent external skyline algorithms are actually better behaved, although they do not have as good an apparent asymptotic complexity. We introduce a new external algorithm, LESS, that combines the best features of these, experimentally evaluate its effectiveness and improvement over the field, and prove its average-case running time is $\mathcal{O}(kn)$.

1 Introduction

The *maximal vector problem* is to find the subset of the vectors such that each is not *dominated* by any of the vectors from the set. One vector dominates another if each of its components has an equal or higher value than the other vector's corresponding component, and it has a higher value on at least one of the corresponding components. One may equivalently consider *points* in a k-dimensional space instead of vectors. In this context, the maximals have also been called the *admissible* points, and the set of maximals called the *Pareto set.* This problem has been considered for many years, as identifying the maximal vectors—or admissible points—is useful in many applications. A number of algorithms have been proposed for efficiently finding the maximals.

The maximal vector problem has been rediscovered recently in the database context with the introduction of *skyline queries.* Instead of vectors or points, this is to find the maximals over *tuples.* Certain columns (with numeric domains) of the input relation are designated as the skyline criteria, and dominance is then defined with respect to these. The non-dominated tuples then constitute the skyline set.

Skyline queries have attracted a fair amount of attention since their introduction in [5]. It is thought that skyline offers a good mechanism for incorporating preferences into relational queries, and, of course, its implementation could enable maximal vector applications to be built on relational database systems efficiently. While the idea itself is older, much of the recent skyline work has focused on designing good algorithms that are well-behaved in the context of a relational query engine and are *external* (that is, that work over data sets too large to fit in main-memory).

On the one hand, intuition says it should be fairly trivial to design a reasonably good algorithm for finding the maximal vectors. We shall see that many approaches are $\mathcal{O}(n)$ average-case running time. On the other hand, performance varies widely for the algorithms when applied to input sets of large, but realistic, sizes (n) and reasonable dimensionality (k). In truth, designing a good algorithm for the maximal vector problem is far from simple, and there are many subtle, but important, issues to attend.

In this paper, we focus on *generic* maximal-vector algorithms; that is, on algorithms for which prepro-

Part of this work was conducted at William & Mary where Ryan Shipley was a student and Parke Godfrey was on faculty while on leave of absence from York.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

cessing steps or data-structures such as indexes are not required. The contributions are two-fold: first, we thoroughly analyze the existing field of generic maximal vector algorithms, especially with consideration of the dimensionality k's impact; second, we present a new algorithm that essentially combines aspects of a number of the established algorithms, and offers a substantial improvement over the field.

In §2, we simultaneously review the work in the area and analyze the proposed algorithms' runtime performances. We first introduce a *cost model* on which we can base average-case analyses (§2.1). This assumes an estimate of the number of maximals expected, on average, assuming statistical independence of the dimensions and distinct values of the vectors along each dimension. We summarize the generic algorithmic approaches—both older algorithms and newer, external skyline algorithms—for computing maximal vector sets (§2.2). We briefly discuss some of the index-based approaches to maximal vector computation, and why index-based approaches are necessarily of limited utility (§2.3). We then formally analyze the run-time performances of the generic algorithms to identify the bottlenecks and compare advantages and disadvantages between the approaches, first the divide-and-conquer approaches (§2.4), and then the external, "skyline" approaches (§2.5).

In §3, we present a new algorithm for maximal vector computation, LESS (*linear elimination sort for skyline*), the design of which is motivated by our analyses and observations (§3.1). We present briefly some experimental evaluation of LESS that demonstrates its improvement over the existing field (§3.2). We formally analyze its runtime characteristics, prove it has $\mathcal{O}(kn)$ average runtime performance, and demonstrate its advantages with respect to the other algorithms (§3.3). Finally, we identify the key bottlenecks for any maximal-vector algorithm, and discuss further ways that LESS could be improved (§3.4).

We discuss future issues and conclude in §4.

2 Algorithms and Analyses

2.1 Cost Model

A simple approach would be to compare each point against every other point to determine whether it is dominated. This would be $\mathcal{O}(n^2)$, for any fixed dimensionality k. Of course, once a point is found that dominates the point in question, processing for that point can be curtailed. So average-case running time should be significantly better, even for this simple approach. In the best-case scenario, for each non-maximal point, we would find a dominating point for it immediately. So each non-maximal point would be eliminated in $\mathcal{O}(1)$ steps. Each maximal point would still be expensive to verify; in the least, it would need to be compared against each of the other maximal points to show it is not dominated. If there are not too many maximals, this will not be too expensive. Given m, the expected number of maximals, if $m < \sqrt{n}$, the number of maximal-to-maximal comparisons, $\mathcal{O}(m^2)$, is $\mathcal{O}(n)$. Thus, assuming m is sufficiently small, in best-case, this approach is $\mathcal{O}(n)$. A goal then is for algorithms with average-case running time of $\mathcal{O}(n)$.

Performance then is also dependent on the number of maximals (m). In worst-case, all points are maximal ($m = n$); that is, no point dominates any other. We shall consider average-case performance based on the expected value of m. To do this, we shall need certain assumptions about the input set.

Definition 1 *Consider the following properties of a set of points:*

a. *(*independence*) the values of the points over a single dimension are statistically independent of the values of the points along any other dimension;*
b. *(*distinct values*) points (mostly) have distinct values along any dimension (that is, there are not many repeated values); and*
c. *(*uniformity*) the values of the points along any one dimension are uniformly distributed.*

Collectively, the properties of independence *and* distinct values *are called* component independence *(CI) [4]. (In some cases, we shall just assume CI as the property of* uniformity *is not necessary for the result at hand.) Let us call collectively the three properties* uniform independence *(UI).*[1]

Additionally, assume under uniformity*, without loss of generality, that any value is on the interval* $(0, 1)$.[2]

Under CI, the expected value of the number of maximals is known [6, 11]: $m = \mathsf{H}_{k-1,n}$, where $\mathsf{H}_{k,n}$ is the k-th order harmonic of n. Let $\mathsf{H}_{0,n} = 1$, for $n > 0$, and $\mathsf{H}_{k,n}$ be inductively defined as $\mathsf{H}_{k,n} = \sum_{i=1}^{n} \frac{\mathsf{H}_{k-1,i}}{i}$, for $k > 1$. $\mathsf{H}_{k,n} \approx \mathsf{H}_{1,n}^{k}/k! \approx ((\ln n) + \gamma)^k/k!$.

When the distinct-value assumption is violated and there are many repeated values along a dimension, the expected value of m goes *down*, up to the point at which the set is dominated by duplicate points (that is, equivalent on all the dimensions) [11].

For best-case, assume that there is a total ordering of the points, $p_1, \ldots, p_n$, such that any p_i dominates all p_j, for $i < j$. Thus, in best-case, $m = 1$ (the one point being p_1).[3]

[1]Note that, given a set of points that is CI, we could replace the points' values with their ordinal ranks over the data set with respect to each dimension. Then the set of points would be UI. However, this transformation would not be computationally insignificant.

[2]This is without loss of generality since it makes no further assumptions about the data distributions. The data values can always be normalized onto $(0, 1)$. Furthermore, this adds no significant computational load. Knowing the maximum and minimum values of the points for each dimension is sufficient to make the mapping in a single pass.

[3]We consider a total order so that, for any subset of the

algorithm		ext.	best-case		average-case		worst-case	
DD&C	[14]	no	$\mathcal{O}(kn\lg n)$	§2.2	$\Omega(kn\lg n + (k-1)^{k-3}n)$	Thm.12	$\mathcal{O}(n\lg^{k-2} n)$	[14]
LD&C	[4]	no	$\mathcal{O}(kn)$	§2.2	$\mathcal{O}(n)$, $\Omega((k-1)^{k-2}n)$	[4], Thm. 11	$\mathcal{O}(n\lg^{k-1} n)$	[4]
FLET	[3]	no	$\mathcal{O}(kn)$	§2.2	$\mathcal{O}(kn)$	[3]	$\mathcal{O}(n\lg^{k-2} n)$	[3]
SD&C	[5]	–	$\mathcal{O}(kn)$	Thm. 2	$\Omega(\sqrt{k}2^{2k}n)$	Thm. 10	$\mathcal{O}(kn^2)$	Thm. 3
BNL	[5]	yes	$\mathcal{O}(kn)$	Thm. 4	–		$\mathcal{O}(kn^2)$	Thm. 5
SFS	[8]	yes	$\mathcal{O}(n\lg n + kn)$	Thm. 6	$\mathcal{O}(n\lg n + kn)$	Thm. 8	$\mathcal{O}(kn^2)$	Thm. 9
LESS	–	yes	$\mathcal{O}(kn)$	Thm. 14	$\mathcal{O}(kn)$	Thm. 13	$\mathcal{O}(kn^2)$	Thm. 15

Figure 1: The generic maximal vector algorithms.

We shall assume that $k \ll n$. Furthermore, we assume that, generally, $k < \lg n$. We include the dimensionality k in our $\mathcal{O}$-analyses.

We are now equipped to review the proposed algorithms for finding the maximal vectors, and to analyze their asymptotic runtime complexities ($\mathcal{O}$'s). Not all of the $\mathcal{O}(n)$ average cases can be considered equivalent without factoring in the impact of the dimensionality k. Furthermore, we are interested in external algorithms, so I/O cost is pertinent. After our initial analyses, we shall look into these details.

2.2 The Algorithms

The main (generic) algorithms that have been proposed for maximal vectors are listed in Figure 1. We have given our own names to the algorithms (not necessarily the same names as used in the original papers) for the sake of discussion. For each, whether the algorithm was designed to be external is indicated, and the known best, average, and worst case running time analyses—with respect to CI or UI and our model for average case in §2.1—are shown. For each runtime analysis, it is indicated where the analysis appears. For each marked with '§', it follows readily from the discussion of the algorithm in that Section. BNL's average-case, marked with '–', is not directly amicable to analysis (as discussed in §2.5). The rest are proven in the indicated theorems.[4]

The first group consists of divide-and-conquer-based algorithms. DD&C (*double divide and conquer*) [14], LD&C (*linear divide and conquer*) [4], and FLET (*fast linear expected time*) [3] are "theoretical" algorithms that were proposed to establish the best bounds possible on the maximal vector problem. No attention was paid to making the algorithms external. Their initial asymptotic analyses make them look attractive, however.

DD&C does divide-and-conquer over both the data (n) *and* the dimensions (k). First, the input set is sorted in k ways, once for each dimension. Then, the sorted set is then split in half along one of the dimensions, say d_k, with respect to the the sorted order over d_k. This is recursively repeated until the resulting set is below threshold in size (say, a single point). At the bottom of this recursive divide, each set (one point) consists of just maximals with respect to that set. Next, these (locally) maximal sets are merged. On each merge, we need to eliminate any point that is not maximal with respect to the unioned set. Consider merging sets $\mathcal{A}$ and $\mathcal{B}$. Let all the maximals in $\mathcal{A}$ have a higher value on dimension d_k than those in $\mathcal{B}$ (given the original set was divided over the sorted list of points with respect to dimension d_k). The maximals of $\mathcal{A} \cup \mathcal{B}$ are determined by applying DD&C, but now over dimensions $d_1, \ldots, d_{k-1}$, so with reduced dimensionality.[5]

Once the dimensionality is three, an efficient special-case algorithm can be applied. Thus, in worst-case, $\mathcal{O}(n\lg^{k-2} n)$ steps are taken. In the best-case, the double-divide-and-conquer is inexpensive since each maximal set only has a single point. (It resolves to $\mathcal{O}(n)$.) However, DD&C needs to sort the data by each dimension initially, and this costs $\mathcal{O}(kn\lg n)$. We establish DD&C's average-case performance in Section 2.4. Note that DD&C establishes that the maximal vector problem is, in fact, $o(n^2)$.

LD&C [4] improves on the average-case over DD&C. Their analysis exploits the fact that they showed m to be $\mathcal{O}(\ln^{k-1} n)$ average-case. LD&C does a basic divide-and-conquer recursion first, randomly splitting the set into two equal sets each time. (The points have not been sorted.) Once a set is below threshold size, the maximals are found. To merge sets, the DD&C algorithm is applied. This can be modeled by the recurrence

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n/2) + (\ln^{k-1} n)\lg^{k-2}(\ln^{k-1} n) \end{aligned}$$

Note that $(\ln^{k-1} n)\lg^{k-2}(\ln^{k-1} n)$ is $o(n)$. Therefore, LD&C is average-case linear, $\mathcal{O}(n)$ [4].

In best case, each time LD&C calls DD&C to merge to maximal sets, each maximal set contains a single point. Only one of the two points survives in the re-

points, there is just one maximal with respect to that subset. This is necessary for discussing the divide-and-conquer-based algorithms.

[4]Some of the theorems are relatively straightforward, but we put them in for consistency.

[5]All points in $\mathcal{A}$ are marked so none will be thrown away. Only points in $\mathcal{B}$ can be dominated by points in $\mathcal{A}$, since those in $\mathcal{A}$ are better along dimension d_k.

sulting maximal set. This requires that DD&C recurse to the bottom of its dimensional divide, which is k deep, to determine the winning point. $\mathcal{O}(n)$ merges are then done at a cost of $\mathcal{O}(k)$ steps each. Thus, LD&C's average-case running time is, at least, $\Omega(kn)$. (In Section 2.4, we establish that, in fact, it is far worse.) In worst case, the set has been recursively divided an extra time, so LD&C is $\lg n$ times worse than DD&C.

FLET [3] takes a rather different approach to improving on DD&C's average-case. Under UI,[6] a virtual point x—not necessarily an actual point in the set—is determined so that the probability that no point from the set dominates it is less than $1/n$. The set of points is then scanned, and any point that is dominated by x is eliminated. It is shown that the number of points x will dominate, on average, converges on n in the limit, and the number it does not is $o(n)$. It is also tracked while scanning the set whether any point is found that dominates x. If some point did dominate x, it does not matter that the points that x dominates were thrown away. Those eliminated points are dominated by a real point from the set anyway. DD&C is then applied to the $o(n)$ remaining points, for a $\mathcal{O}(kn)$ average-case running time. This happens at least $(n-1)/n$ fraction of trials. In the case no point was seen to dominate x, which should occur less than $1/n$ fraction of trials, DD&C is applied to the whole set. However, DD&C's $\mathcal{O}(n\lg^{k-2} n)$ running time in this case is amortized by $1/n$, and so contributes $\mathcal{O}(\lg^{k-2} n)$, which is $o(n)$. Thus, the amortized, average-case running time of FLET is $\mathcal{O}(kn)$. FLET is no worse asymptotically than DD&C in worst case.

FLET's average-case runtime is $\mathcal{O}(kn)$ because FLET compares $\mathcal{O}(n)$ number of points against point x. Each comparison involves comparing all k components of the two points, and so is k steps. DD&C and LD&C never compare two points directly on all k dimensions since they do divide-and-conquer also over the dimensions. In [4] and [14], DD&C and LD&C were analyzed with respect to a fixed k. We are interested in how k affects their performance, though.

The second group—the *skyline* group—consists of external algorithms designed for skyline queries. Skyline queries were introduced in [5] along with two general algorithms proposed for computing the skyline in the context of a relational query engine.

The first general algorithm in [5] is SD&C, *single divide-and-conquer*. It is a divide-and-conquer algorithm similar to DD&C and LD&C. It recursively divides the data set. Unlike LD&C, DD&C is not called to merge the resulting maximal sets. A divide-and-conquer is not performed over the dimensions. Consider two maximal sets $\mathcal{A}$ and $\mathcal{B}$. SD&C merges them by comparing each point in $\mathcal{A}$ against each point in $\mathcal{B}$, and vice versa, to eliminate any point in $\mathcal{A}$ dominated by a point in $\mathcal{B}$, and vice versa, to result in just the maximals with respect to $\mathcal{A} \cup \mathcal{B}$.

[6]For the analysis of FLET, we need to make the additional assumption of *uniformity* from Def. 1.

Theorem 2 *Under CI (Def. 1) and the model in §2.1,* SD&C *has a best-case runtime of* $\mathcal{O}(kn)$.

Proof 2 *Let* $m_{\mathcal{A}}$ *denote the number of points in* $\mathcal{A}$ *(which are maximal with respect to* $\mathcal{A}$*). Let* $m_{\mathcal{A}\setminus\mathcal{B}}$ *denote the number of points in* $\mathcal{A}$ *that are maximal with respect to* $\mathcal{A}\cup\mathcal{B}$*. Likewise, define* $m_{\mathcal{B}}$ *and* $m_{\mathcal{B}\setminus\mathcal{A}}$ *in the same way with respect to* $\mathcal{B}$*. An upper bound on the cost of merging* $\mathcal{A}$ *and* $\mathcal{B}$ *is* $km_{\mathcal{A}}m_{\mathcal{B}}$ *and a lower bound is* $km_{\mathcal{A}\setminus\mathcal{B}}m_{\mathcal{B}\setminus\mathcal{A}}$*. In best case,* SD&C *is* $\mathcal{O}(kn)$. □

For a fixed k, average case is $\mathcal{O}(n)$. (We shall consider more closely the impact of k on the average case in §2.4.)

Theorem 3 *Under CI (Def. 1),* SD&C *has a worst-case runtime of* $\mathcal{O}(kn^2)$.

Proof 3 *The recurrence for* SD&C *under worst case is*

$$T(1) = 1$$
$$T(n) = 2T(n/2) + (n/2)^2$$

This is $\mathcal{O}(n^2)$ *number of comparisons. Each comparison under* SD&C *costs k steps, so the runtime is* $\mathcal{O}(kn^2)$. □

No provisions were made to make SD&C particularly well behaved relationally, although it is clearly more amenable to use as an external algorithm than DD&C (and hence, LD&C and, to an extent, FLET too, as they rely on DD&C). The divide stage of SD&C is accomplished trivially by bookkeeping. In the merge stage, two files, say $\mathcal{A}$ and $\mathcal{B}$, are read into main memory, and their points pairwise compared. The result is written out. As long as the two input files fit in main memory, this works well. At the point at which the two files are too large, it is much less efficient. A block-nested loops strategy is employed to compare all $\mathcal{A}$'s points against all of $\mathcal{B}$'s, and vice versa.

The second algorithm proposed in [5] is BNL, *block nested loops*. This is basically an implementation of the simple approach discussed in §2.1, and works remarkably well. A *window* is allocated in main memory for collecting points (tuples). The input file is scanned. Each point from the input stream is compared against the window's points. If it is dominated by any of them, it is eliminated. Otherwise, any window points dominated by the new point are removed, and the new point itself is added to the window.

At some stage, the window may become full. Once this happens, the rest of the input file is processed differently. As before, if a new point is dominated by a window point, it is eliminated. However, if the new point is not dominated, it is written to an overflow file. (Dominated window points are still eliminated as before.) The creation of an overflow file means another *pass* will be needed to process the overflow points. Thus, BNL is a multi-pass algorithm. On a subsequent pass, the previous overflow file is read as the input. Appropriate bookkeeping tracks when a

window point has gone full cycle. (That is, it has been compared against all currently surviving points.) Such window points can be removed from the window and written out, or pipelined along, as maximals.

BNL differs substantially from the divide-and-conquer algorithms. As points are continuously replaced in the window, those in the window are a subset of the maximals with respect to the points seen so far (modulo those written to overflow). These global maximals are much more effective at eliminating other points than are the local maximals computed at each recursive stage in divide-and-conquer.

Theorem 4 *Under CI (Def. 1) and the model in §2.1,* BNL *has a best-case runtime of* $\mathcal{O}(kn)$.

Proof 4 BNL*'s window will only ever contain one point. Each new point off the stream will either replace it or be eliminated by it. Thus* BNL *will only require one pass.* □

A good average case argument with respect to UI is difficult to make. We discuss this in §2.5. Let w be the size limit of the window in number of points.

Theorem 5 *Under CI (Def. 1),* BNL *has a worst-case runtime of* $\mathcal{O}(kn^2)$.

Proof 5 *In worst case, every point will need to be compared against every other point for* $\mathcal{O}(kn^2)$. *This requires* $\lceil n/w \rceil$ *passes. Each subsequent overflow file is smaller by w points. So this requires writing* $n^2/2w$ *points and reading* $n^2/2w$ *points. The size of w is fixed. In addition to requiring* $\mathcal{O}(n^2)$ *I/O's, every record will need to be compared against every other record. Every record is added to the window; none is ever removed. Each comparison costs k steps. So the work of the comparisons is* $\mathcal{O}(kn^2)$. □

In [8], SFS, *sort filter skyline*, is presented. It differs from BNL in that the data set is topologically sorted initially. A common nested sort over the dimensions $d_1, \ldots, d_k$, for instance, would suffice. In [9], the utility of sorting for finding maximals and SFS are considered in greater depth. Processing the sorted data stream has the advantage that no point in the stream can be dominated by any point that comes after it. In [8, 9], sorting the records by *volume* descending, $\prod_{i=1}^{k} t[d_i]$;[7] or, equivalently, by *entropy* descending, $\sum_{i=1}^{k} \ln t[d_i]$ (with the assumption that the values $t[d_i] > 0$ for all records t and dimensions i) is advocated.[8] This has the advantage of tending to push records that dominate many records towards the beginning of the stream.

SFS sorts by volume because records with higher volumes are more likely to dominate more records in the set. They have high "dominance" numbers. Assuming UI, the number of records a given record dominates is proportional to its volume. By putting these earlier in the stream, non-maximal records are eliminated in fewer comparisons, on average. The importance of this effect is emphasized in the discussion of LESS in §3 and in the proof that LESS is $\mathcal{O}(kn)$ (Thm. 13).

SFS maintains a main-memory window as does BNL. It is impossible for a point off the stream to dominate any of the points in the window, however. Any point is thus known to be maximal at the time it is placed in the window. The window's points are used to eliminate stream points. Any stream point not eliminated is itself added to the window. As in BNL, once the window becomes full, surviving stream points must be written to an overflow file. At the end of the input stream, if an overflow file was opened, another pass is required. Unlike BNL, the window can be cleared at the beginning of each pass, since all points have been compared against those maximals. The overflow file is then used as the input stream.

SFS has less bookkeeping overhead than BNL since when a point is added to the window, it is already known that the point is maximal. This also means that SFS is progressive: at the time a point is added to the window, it can also be shipped as a maximal to the next operation. In [8], it was shown that SFS performs better I/O-wise than BNL, and runs in better time (and this *includes* SFS's necessary sorting step). The experiments in [8] were run over million-tuple data sets and with dimensions of five to seven.

Theorem 6 *Under CI (Def. 1) and the model in §2.1,* SFS *has a best-case runtime of* $\mathcal{O}(kn + n\lg n)$.

Proof 6 *Under our best-case scenario, there is one maximal point. This point must have the largest volume. Thus it will be the first point in* SFS*'s sorted stream, and the only point to be ever added to the window. This point will eliminate all others in one pass. So* SFS *is sorting plus* $\mathcal{O}(kn)$ *in best-case, and works in one filtering pass.* □

For average-case analysis of SFS, we need to know how many of the maximal points dominate any given non-maximal. For any maximal point, it will be compared against every maximal point before it in the sorted stream to confirm its maximality. Thus there will be $m^2/2$ of these comparisons. For any non-maximal point, how many maximals (points in the window) will it need to be compared against before being eliminated?

Lemma 7 *Under UI (Def. 1), in the limit of n, the probability that any non-maximal point is dominated by the maximal point with the highest volume converges to one.*

Proof 7 *Assume UI. Let the values of the points be distributed uniformly on* $(0, 1)$ *on each dimension.*

We draw on the proof of FLET*'s average case runtime in [3]. Consider the (virtual) point x with coordinates* $x[d_i] = 1 - ((\ln n)/n)^{1/k}$, *for each* $i \in 1, \ldots, k$. *The probability that no point from the data set dominates x then is* $(1 - (\ln n)/n)^n$, *which is at most* $e^{-\ln n} = 1/n$.

[7] As in the discussion about FLET, we are assuming *uniformity* (Def. 1.

[8] Keeping entropy instead of volume avoids register overflow.

The expected number of points dominated by x (and hence, dominated by any point that dominates x) is $(1 - ((\ln n)/n)^{1/k})^k$.

$$\lim_{n\to\infty} (1 - ((\ln n)/n)^{1/k})^k = 1$$

Thus any maximal with a volume greater than x's (which would include any points that dominate x) will dominate all points in the limit of n. The probability there is such a maximal is greater than $(n-1)/n$*, which converges to one in the limit of n.* □

Theorem 8 *Under UI (Def. 1),* SFS *has an average-case runtime of* $\mathcal{O}(kn + n\lg n)$.

Proof 8 *The sort phase for* SFS *is* $\mathcal{O}(n\lg n)$. *On the initial pass, the volume of each point can be computed at* $\mathcal{O}(kn)$ *expense. During the filter phase of* SFS, $m^2/2$ *maximal-to-maximal comparisons will be made. Expected m is* $\Theta((\ln^{k-1} n)/(k-1)!)$*, so this is* $o(n)$. *Number of comparisons of non-maximal to maximal is* $\mathcal{O}(n)$. *Thus the comparison cost is* $\mathcal{O}(kn)$. □

Theorem 9 *Under CI (Def. 1),* SFS *has a worst-case runtime of* $\mathcal{O}(kn^2)$.

Proof 9 *In the worst-case, all records are maximal. Each record will be placed in the skyline window after being compared against the records currently there. This results in* $n(n-1)/2$ *comparisons, each taking k steps. The sorting phase is* $\mathcal{O}(n\lg n)$ *again.* □

On the one hand, it is observed experimentally that SFS makes fewer comparisons than BNL. SFS compares only against maximals, whereas BNL will often have non-maximal points in its window. On the other hand, SFS does require sorting. For much larger n, the sorting cost will begin to dominate SFS's performance.

2.3 Index-based Algorithms and Others

In this paper, as stated already, we focus on generic algorithms to find the maximal vectors, algorithms that do not require any pre-processing or pre-existing data-structures. Any query facility that is to offer maximal vector, or skyline, computation would require a generic algorithm for those queries for which the pre-existing indexes are not adequate.

There has been a good deal of interest though in index-based algorithms for skyline. The goals are to be able to evaluate the skyline without needing to scan the entire dataset—so for sub-linear performance, $o(n)$—and to produce skyline points *progressively*, to return initial answers as quickly as possible.

The *shooting-stars* algorithm [13] exploits R-trees and modifies nearest-neighbors approaches for finding skyline points progressively. This work is extended upon in [15] in which they apply branch-and-bound techniques to reduce significantly the I/O overhead. In [10, 12], bitmaps are explored for skyline evaluation, appropriate when the number of values possible along a dimension is small. In [1], an algorithm is presented as instance-optimal when the input data is available for scanning in k sorted ways, sorted along each dimension. If a tree index were available for each dimension, this approach could be applied.

Index-based algorithms for computing the skyline (the maximal vectors) additionally have serious limitations. The performance of indexes—such as R-trees as used in [13, 15]—does not scale well with the number of dimensions. Although the dimensionality of a given skyline query will be typically small, the *range* of the dimensions over which queries can be composed can be quite large, often exceeding the performance limit of the indexes. For an index to be of practical use, it would need to cover most of the dimensions used in queries.

Note also that building several indexes on small subsets of dimensions (so that the union covers all the dimensions) does not suffice, as the skyline of a set of dimensions cannot be computed from the skylines of the subsets of its dimensions. It is possible, and probable, that

$$\mathsf{maxes}_{\{\mathsf{d}_1,\ldots,\mathsf{d}_i\}}(\mathbf{T}) \cup \mathsf{maxes}_{\{\mathsf{d}_{i+1},\ldots,\mathsf{d}_k\}}(\mathbf{T}) \subsetneq \mathsf{maxes}_{\{\mathsf{d}_1,\ldots,\mathsf{d}_k\}}(\mathbf{T})$$

Furthermore, if the distinct-values assumption from §2.1 is lifted, the union is no longer even guaranteed to be a subset. (This is due to the possibility of ties over, say, $\mathsf{d}_1, \ldots, \mathsf{d}_i$.)

Another difficulty with the use of indexes for computing skyline queries is the fact that the skyline operator is *holistic*, in the sense of holistic aggregation operators. The skyline operator is not, in general, commutative with selections.[9] For any skyline query that involves a select condition then, an index that would have applied to the query without the select will not be applicable.

Finally, in a relational setting, it is quite possible that the input set—for which the maximals are to be found—is itself computed via a sub-query. In such a case, there are no available indexes on the input set.

2.4 The Case against Divide and Conquer

Divide-and-conquer algorithms for maximal vectors face two problems:

1. it is not evident how to make an efficient external version; and,
2. although the asymptotic complexity with respect to n is good, the multiplicative "constant"—and the effect of the dimensionality k—may be bad.

Since there are algorithms with better average-case run-times, we would not consider DD&C. Furthermore, devising an effective external version for it seems impossible. In DD&C, the data set is sorted first in k ways, once for each dimension. The sorted orders could be implemented in main memory with one node per point and a linked list through the nodes for each dimension. During the merge phase, DD&C does not

[9] In [7], cases of commutativity of skyline with other relational operators are shown.

re-sort the data points; rather, the sorted orders are maintained. In a linked-list implementation, it is easy to see how this could be done. It does not look possible to do this efficiently as an external algorithm.

LD&C calls DD&C repeatedly. Thus, for the same reasons, it does not seem possible to make an effective external version of LD&C. FLET calls DD&C just once. Still, since the number of points that remain after FLET's initial scan and elimination could be significant, FLET would also be hard to externalize.

SD&C was introduced in [5] as a viable external divide-and-conquer for computing maximal vectors. As we argued above, and as is argued in [15], SD&C is still far from ideal as an external algorithm. Furthermore, its runtime performance is far from what one might expect.

Each merge that SD&C performs of sets, say, $\mathcal{A}$ and $\mathcal{B}$, every maximal with respect to $\mathcal{A} \cup \mathcal{B}$ that survives from $\mathcal{A}$ must have been compared against every maximal that survives from $\mathcal{B}$, and vice-versa. This is a floor on the number of comparisons done by the merge. We know the number of maximals in average case under CI. Thus we can model SD&C's cost via a recurrence. The expected number of maximals out of n points of k dimensions under CI is $\mathsf{H}_{k-1,n}$; $(\ln^{k-1} n)/(k-1)!$ converges on this from below, so we can use this in a floor analysis.

Theorem 10 *Under CI (Def. 1),* SD&C *has average-case runtime of* $\Omega(\sqrt{k}\,2^{2k} n)$.

Proof 10 *Let* $n = 2^q$ *for some positive integer* q*, without loss of generality. Consider the function* T *as follows.*

$$
\begin{aligned}
T(1) &= 1\\
T(n) &= 2T(n/2) + (\tfrac{1}{2}(\ln^{k-1} n)/(k-1)!)^2\\
&\qquad c_1 = 1/(4(k-1)!^2) \qquad D = 2k-2\\
&= 2T(n/2) + c_1 \ln^D n\\
&= c_1 \sum_{i=1}^{q} 2^i (\ln n - \ln 2^i)^D\\
&\qquad c_2 = c_1/(\lg_2 e)^D\\
&= c_2 \sum_{i=1}^{q} 2^i (\lg_2 n - \lg_2 2^i)^D\\
&= c_2 \sum_{i=1}^{q} 2^i (q-i)^D = c_2 \sum_{i=0}^{q-1} 2^{q-i} i^D\\
&\qquad \sum_{i=0}^{j} 2^{j-i} i^D \approx (\lg_2 e)^{D-1} D!\, 2^{j+1}\\
&\approx c_2 (\lg_2 e)^{D-1} D!\, 2^q = \tfrac{1}{4}(\ln 2)\tbinom{2k-2}{k-1} n\\
&\qquad \tbinom{2j}{j} \approx 2^{2j}/\sqrt{\pi j} \text{ (by Stirling's approximation)}\\
&\approx \frac{\ln 2}{\sqrt{\pi(k-1)}} 2^{2k-4} n
\end{aligned}
$$

This counts the number of comparisons. Each comparison costs k *steps.*

For each merge step, we assume that the expected value of maximals survive, and that exactly half came from each of the two input sets. In truth, fewer might come from $\mathcal{A}$ *and more from* $\mathcal{B}$ *sometimes. So the square of an even split is an over-estimate, given variance of resulting set sizes. In [2], it is established that the variance of the number of maximals under CI converges on* $\mathsf{H}_{k-1,n}$*. Thus in the limit of* n*, runtime of* SD&C *will converge up to an asymptotic above the recurrence.* □

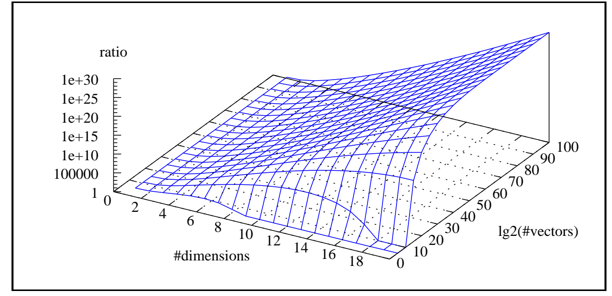


Figure 2: Behavior of LD&C.

This is bad news. SFS requires n comparisons in the limit of n, for any fixed dimensionality k. SD&C, however, requires on the order of $(2^{2k}/\sqrt{k}) \times n$ comparisons in n's limit!

In [5], it was advocated that SD&C is more appropriate for larger k (say, for $k > 7$) than BNL, and is the preferred solution for data sets with large k. Our analysis conclusively shows the opposite: SD&C will perform increasingly worse for larger k and with larger n. We believe their observation was an artifact of the experiments in [5]. The data sets they used were only 100,000 points, and up to 10 dimensions. Even if the data sets used were a million points instead (and 10 dimensions), SD&C would have performed proportionally significantly much worse.

We can model LD&C's behavior similarly. For a merge (of $\mathcal{A}$ and $\mathcal{B}$) in LD&C, it calls DD&C. Since $\mathcal{A}$ and $\mathcal{B}$ are maximal sets, most point will survive the merge. The cost of the call to DD&C is bounded below by its worst-case runtime over the number of points that survive. The double recursion must run to complete depth for these. So if m points survive the merge, the cost is $m \lg_2^{k-2} m$ steps. As in the proof of Thm. 10 for SD&C, we can approximate the expected number of maximals from below. Let $m_{k,n} = (\ln^{k-1}(n+\gamma))/(k-1)!$. The recurrence is

$$
\begin{aligned}
T(1) &= 1\\
T(n) &= 2T(n/2) + \max(m_{k,n} \lg_2^{k-2} m_{k,n}, 1)
\end{aligned}
$$

We plot this in Figure 2.[10] This shows the *ratio* of the number of comparisons over n. The recurrence asymptotically converges to a constant value for any given k. It is startling to observe that the k-overhead of LD&C appears to be worse than that of SD&C! The explanation is that $m_{k,i} \lg_2^{k-2} m_{k,i}$ is larger initially than

[10]The behavior near the *dimensions* axis is an artifact of our log approximation of $\mathsf{H}_{k-1,i}$, the expected number of maximals. In computing the graph, $m_{k,i} \lg_2^{k-2} m_{k,i}$ is rounded up to one whenever it evaluates to less than one.

is $m_{k,i}^2$, for the small i sizes of data sets encountered near the bottom of the divide-and-conquer. (Of course $m_{k,i}^2 \gg m_{k,i} \lg_2^{k-2} m_{k,i}$ in i's limit; or, in other words, as i approaches n each subsequent merge level, for very large n.) However, it is those initial merges near the bottom of the divide-and-conquer that contribute most to the cost overall, since there are many more pairs of sets to merge at those levels. Next, we prove a lower bound on LD&C's average case.

Theorem 11 *Under CI (Def. 1),* LD&C *has average-case runtime of* $\Omega((k-1)^{k-2}n)$.

Proof 11 *Let $n = 2^q$ for some positive integer q, without loss of generality.*

$$
\begin{aligned}
& m_{k,n} \lg_2^{k-2} m_{k,n} \\
& \approx ((\ln^{k-1} n)/(k-1)!) \lg_2 (((\ln^{k-1} n)/(k-1)!))^{k-2} \\
& \qquad c_1 = 1/(k-1)! \\
& = c_1 (\ln 2)^{k-1} (\lg_2 n)^{k-1} \\
& \qquad \cdot (\lg_2((\ln 2)^{k-1}(\lg_2 n)^{k-1}) - \lg_2 (k-1)!)^{k-2} \\
& \qquad c_2 = \ln^{k-1} 2 \\
& > c_1 c_2 (\lg_2^{k-1} n) \\
& \qquad \cdot ((k-1)((\lg_2 q) + (\ln 2) - (\lg_2 (k-1)))^{k-2} \\
& \qquad \textit{when } \lg_2 q > \lg_2 (k-1) \\
& > c_1 c_2 (\lg_2^{k-1} n)(k-1)^{k-2} \\
& \qquad \textit{when } \lg_2 q - \lg_2 (k-1) \geq 1 \\
& \qquad \textit{thus } q \geq 2(k-1)
\end{aligned}
$$

Let $l = 2(k-1)$. Consider the function T as follows.

$$
\begin{aligned}
T(n) &= 2T(n/2) + \mathsf{max}(m_{k,n} \lg_2^{k-2} m_{k,n}, 1) \\
& > c_1 c_2 (k-1)^{k-2} \sum_{i=l}^{q} 2^{q-i} i^{k-1} \\
& \qquad \textit{for } n \geq 2^l \\
& = c_1 c_2 (k-1)^{k-2} 2^l \sum_{i=0}^{(q-l)} 2^{(q-l)-i} i^{k-1} \\
& \approx c_1 c_2 (k-1)^{k-2} 2^l (\lg_2^{k-1} e)(k-1)! 2^{(q-l)} \\
& = (k-1)^{k-2} 2^q \\
& = (k-1)^{k-2} n
\end{aligned}
$$

$T(n)$ is a strict lower bound on the number of comparisons that LD&C *makes, in average case. We only sum $T(n)$ for $n \geq 2^l$ and show $T(n) > (k-1)^{k-2} n$.* □

We can use the same reasoning to obtain an asymptotic lower bound on DD&C's average-case runtime.

Theorem 12 *Under CI (Def. 1),* DD&C *has an average-case runtime of* $\Omega(kn \lg n + (k-1)^{k-3} n)$.

Proof 11 DD&C *first does a divide and conquer over the data on the first dimension. During a merge step of this divide-and-conquer, it recursively calls* DD&C *to do the merge, but considering one dimension fewer. The following recurrence provides a lower bound.*

$$
\begin{aligned}
T(n) &= 1 \\
T(n) &= 2T(n/2) + \mathsf{max}(m_{k,n} \lg_2^{k-3} m_{k,n}, 1)
\end{aligned}
$$

By the same proof steps as in the proof for Thm. 11, we can show $T(n) > (k-1)^{k-3} n$. Of course, DD&C *sorts the data along each dimension before it commences divide-and-conquer. The sorting costs* $\Theta(kn \lg n)$. *Thus,* DD&C *considered under CI has average-case runtime of* $\Omega(kn \lg n + (k-1)^{k-3} n)$. □

Should one even attempt to adapt a divide-and-conquer approach to a high-performance, external algorithm for maximal vectors? Divide-and-conquer is quite elegant and efficient in other contexts. We have already noted, however, that it is quite unclear how one could externalize a divide-and-conquer approach for maximal vectors effectively. Furthermore, we believe their average-case run-times are so bad, in light of the dimensionality k, that it would not be worthwhile.

Divide-and-conquer has high overhead with respect to k because the ratio of the number of maximals to the size of the set for a small set is much greater than for a large set. Vectors are not eliminated quickly as they are compared against *local* maximals. The scan-based approaches such as BNL and SFS find *global* maximals—maximals with respect to the entire set—early, and so eliminate non-maximals more quickly.

2.5 The Case against the Skyline Algorithms

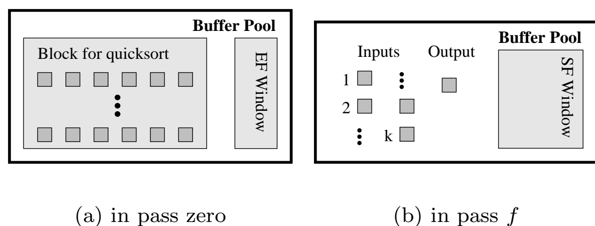
SFS needs to sort initially, which gives it too high an average-case runtime. However, it was shown in [8] to perform better than BNL. Furthermore, it was shown in [8] that BNL is ill-behaved relationally. If the data set is already ordered in some way (but not for the benefit of finding the maximals), BNL can perform very badly. Of course, SFS is immune to input order since it must sort. When BNL is given more main-memory allocation—and thus, its window is larger—its performance deteriorates. This is because any maximal point is necessarily compared against all the points in the window. There no doubt exists an optimal window-size for BNL for a data set of any given n and k. However, not being able to adjust the window size freely means one cannot reduce the number of passes BNL takes.

All algorithms we have observed are CPU-bound, as the number of comparisons to be performed often dominates the cost. SFS has been observed to have a lower CPU-overhead than BNL, and when the window-size is increased for SFS, its performance improves. This is because all points must be compared against all points in the window (since they are maximals) eventually anyway. The number of maximal-to-maximal comparisons for SFS is $m^2/2$ because each point only needs to be compared against those before it in the stream. BNL of course has this cost too, and also compares eventual maximals against additional non-maximals along the way. Often, m is reasonably large, and so this cost is substantial.

3 The LESS Algorithm

3.1 Description

We devise an external, maximal-vector algorithm that we call LESS (*linear elimination sort for skyline*) that combines aspects of SFS, BNL, and FLET, but that



(a) in pass zero

(b) in pass f

Figure 3: Buffer pool for LESS.

does not contain any aspects of divide-and-conquer. LESS filters the records via a *skyline-filter* (SF) window, as does SFS. The record stream must be in sorted order by this point. Thus LESS must sort the records initially too, as does SFS. LESS makes two major changes:

1. it uses an *elimination-filter* (EF) window in pass zero of the external sort routine to eliminate records quickly; and
2. it combines the final pass of the external sort with the first skyline-filter (SF) pass.

The external sort routine used to sort the records is integrated into LESS. Let b be the number of buffer-pool frames allocated to LESS. Pass zero of the standard external sort routine reads in b pages of the data, sorts the records across those b pages (say, using quicksort), and writes the b sorted pages out as a b-length sorted run. All subsequent passes of external sort are *merge* passes. During a merge pass, external sort does a number of $(b-1)$-way merges, consuming all the runs created by the previous pass. For each merge, (up to) $b-1$ of the runs created by the previous pass are read in one page at a time, and written out as a single sorted run.

LESS sorts the records by their entropy scores, as discussed in §2.2 with regards to SFS. LESS additionally eliminates records during pass zero of its external-sort phase. It does this by maintaining a small elimination-filter window. Copies of the records with the best entropy scores seen so far are kept in the EF window (Fig. 3(a)). When a block of records is read in, the records are compared against those in the EF window. Any input record that is dominated by any EF record is dropped. Of the surviving input records, the one with the highest entropy is found. Any records in the EF window that are dominated by this highest entropy record are dropped. If the EF window has room, (a copy of) the input record is added. Else, if the EF window is full but there is a record in it with a lower entropy than this input record, the input record replaces it in the window. Otherwise, the window is not modified.

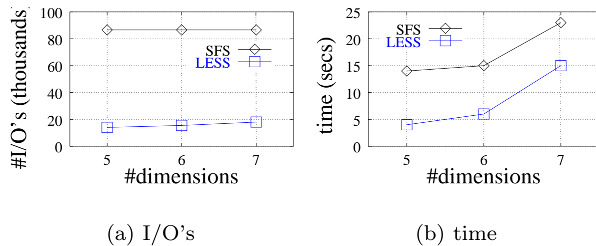
The EF window acts then similarly to the elimination window used by BNL. The records in the EF window are accumulated from the entire input stream. They are not guaranteed to be maximals, of course, but as records are replaced in the EF window, the collection has records with increasingly higher entropy scores. Thus the collection performs well to eliminate other records.

LESS's merge passes of its external-sort phase are the same as for standard external sort, except for the last merge pass. Let pass f be the last merge pass. The final merge pass is combined with the initial skyline-filter pass. Thus LESS creates a skyline-filter window (like SFS's window) for this pass. Of course, there must be room in the buffer pool to perform a multi-way merge over all the runs from pass $f-1$ *and* for a SF window (Fig. 3(b)). As long as there are fewer than $B-2$ runs, this can be done: one frame per run for input, one frame for accumulating maximal records as found, and the rest for the SF window. (If not, another merge pass has to be done before commencing the SF passes.) This is the same optimization done in the standard two-pass sort-merge join, implemented by many database systems. This saves a pass over the data by combining the last merge pass of external sort with join-merge pass. For LESS, this typically saves a pass by combining the last merge pass of the external sort with the first SF pass.

As with SFS, multiple SF passes may be needed. If the SF window becomes full, then an overflow file will be created. Another pass then is needed to process the overflow file. After pass f—if there is an overflow file and thus more passes are required—LESS can allocate $b-2$ frames of its buffer-pool allocation to the SF window for the subsequent passes.

In effect, LESS has all of SFS's benefits with no additional disadvantages. LESS should consistently perform better than SFS. Some buffer-pool space is allocated to the EF window in pass zero for LESS which is not for SFS. Consequently, the initial runs produced by LESS's pass zero are smaller than SFS's; this may occasionally force that LESS will require an additional pass to complete the sort. Of course LESS saves a pass since it combines the last sort pass with the first skyline pass.

LESS also has BNL's advantages, but effectively none of its disadvantages. BNL has the overhead of tracking when window records can be promoted as known maximals. LESS does not need this. Maximals are identified more efficiently once the input is effectively sorted. Thus LESS has the same advantages as does SFS in comparison to BNL. LESS will drop many records in pass zero via use of the EF window. The EF window works to the same advantage as BNL's window. All subsequent passes of LESS then are over much smaller runs. Indeed, LESS's efficiency rests on how effective the EF window is at eliminating records early. In §3.3, we show this elimination is very effective—as it is for FLET and much for the same reason—enough to reduce the sort time to $\mathcal{O}(n)$.



(a) I/O's (b) time

Figure 4: Performance of LESS versus SFS.

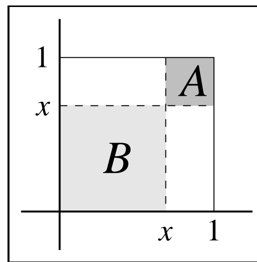


Figure 5: Choosing point v.

3.2 Experimental Evaluation

The LESS prototype is in C and uses Pthreads to implement non-blocking reads and writes. It implements external sort with double buffering. It was implemented and run on RedHat Linux 7.3 on an Intel Pentium III 733 MHz machine.

All tests calculate the skyline of a 500,000 record set with respect to 5, 6, and 7 dimensions. Each record is 100 bytes. A disk-page size of 4,096 bytes is used. Each column used as a skyline criterion has a value $1 \ldots 10,000$. The values were chosen randomly, and the record sets obey the UI criteria from Def. 1.

Input and output is double-buffered to increase performance and to simulate a commercial-caliber relational algorithm. Each input / output block has a thread watchdog that handles the reading / writing of the block. The thread blocks on write but the main process is free to continue processing.

If the EF window is too large, LESS will take more time simply as management of the EF window starts to have an impact. If the EF window is too small (say a single page), the algorithm may become less effective at eliminating records early. As more records survive the sort to the SF-phase, LESS's performance degrades. We experimented with varying the size of the EF window from one to thirty pages. Its size makes virtually no difference to LESS's performance in time or I/O usage. (We make clear why this should be the case in §3.3.) Below five pages, there was some modest increase in LESS's I/O usage. We set the EF window at five pages for what we report here.

We experimented with various buffer pool allotments from 10 to 500 pages. The size affects primarily the efficiency of the sorting phase, as expected. We set the allocation at 76 pages for what we report here.

SFS and BNL are bench marked in [8] where SFS is shown to provide a distinct improvement. Hence we bench marked LESS against SFS. We implemented SFS within our LESS prototype. In essence, when in SFS-mode, the external sort is done to completion *without* using an EF window for elimination, and then the SF passes are commenced.

I/O performance for SFS and LESS are shown in Fig. 4(a). It is the same for SFS in this case for the five, six, and seven dimensional runs. This is because the external sorting is the same in each case, and the number of SF pass I/O's needed were the same. LESS shows a remarkable improvement in I/O usage, as we expect. Many records are eliminated by the EF window during the sorting phase. The I/O usage climbs slightly as the dimensionality increases. This is due to that the elimination during sorting becomes less effective (for a fixed n) as the dimensionality k increases.

The time performance for SFS and LESS are shown in Fig. 4(b). For five dimensions, LESS clocks in at a third of SFS's time. The difference between LESS and SFS closes as the dimensionality increases. This is because, for higher dimensionality, more time is spent by LESS and SFS in the skyline-filtering phase. This is simply due to the fact that more records are maximal. The algorithms become CPU-bound as most of the time is spent comparing skyline records against one another to verify that each is, in fact, maximal.

3.3 Analysis

LESS also incorporates implicitly aspects of FLET. Unlike FLET, we do not want to guess a virtual point to use for elimination. In the rare occasion that the virtual point was not found to be dominated, FLET must process the entire data set by calling DD&C. Such hit-or-miss algorithms are not amenable to relational systems. Instead, LESS uses real points accumulated in the EF window for eliminating. We shall show that these collected points ultimately do as good a job of elimination as does FLET's virtual point. Furthermore, the EF points are points from the data set, so there is no danger of failing in the first pass, as there is with FLET.

To prove that the EF points are effective at eliminating most points, we can construct an argument similar to that used in [3] to prove FLET's $\mathcal{O}(n)$ average-case runtime performance and in Lemma 7.

Theorem 13 *Under UI (Def. 1),* LESS *has an average-case runtime of* $\mathcal{O}(kn)$*.*

Proof 13 *Let the data set be distributed on* $(0,1)^k$ *under UI.*

Consider a virtual point v with coordinate $x \in (0,1)$ on each dimension. Call the "box" of space that dominates v $\mathcal{A}$, and the "box" of space dominated by v $\mathcal{B}$. (This is shown in Fig. 5 for $k = 2$.) The size of $\mathcal{B}$ is then x^k, and the size of $\mathcal{A}$ is $(1-x)^k$. Let $x =$

$(1 - n^{-1/2k})$. Thus the size of $\mathcal{B}$, x^k, is $(1 - n^{-1/2k})^k$. In the limit of n, the size of $\mathcal{B}$ is 1.

$$\lim_{n\to\infty}(1 - n^{-1/2k})^k = 1$$

If a point exists in $\mathcal{A}$, it dominates all points in $\mathcal{B}$. The expected number of points that occupy $\mathcal{A}$ is proportional to $\mathcal{A}$'s volume, which is $1/\sqrt{n}$ by our construction. There are n points, thus $\sqrt{n}$ is the expected number of points occupying $\mathcal{A}$.

If points are drawn at random with replacement from the data set, how many must be explored, on average, before finding one belonging to $\mathcal{A}$?[11] If there were exactly $\sqrt{n}$ points in $\mathcal{A}$, the expected number of draws would be $n/\sqrt{n} = \sqrt{n}$.

Of course, $\sqrt{n}$ is only the expected number of points occupying $\mathcal{A}$. Sometimes fewer than $\sqrt{n}$ points fall in $\mathcal{A}$; sometimes, more. The actual number is distributed around $\sqrt{n}$ via a binomial distribution. Taking the reciprocal of this distribution, the number of draws, on average, to finding a point in $\mathcal{A}$ (or to find no point is in $\mathcal{A}$) is bound above by $(\ln n)\sqrt{n}$.

So during LESS's pass zero, in average case, the number of points that will be processed before finding an $\mathcal{A}$ point is bounded above by $(\ln n)\sqrt{n}$. Once found, that $\mathcal{A}$ point will be added to the EF window; else, there is a point in the EF window already that has a better volume score than this $\mathcal{A}$ point. After this happens, every subsequent $\mathcal{B}$ point will be eliminated.

The number of points that remain, on average, after pass zero then is at most $1 - (1 - n^{-1/2k})^k + (\ln n)\sqrt{n}$. This is $o(n)$. Thus, the surviving set is bound above by n^f, for some $f < 1$. Effectively, LESS only spends effort to sort these surviving points, and $n^f \lg n^f$ is $\mathcal{O}(n)$.

Thus the sort phase of LESS is $\mathcal{O}(kn)$. The skyline phase of LESS is clearly bound above by SFS's average-case, minus the sorting cost. SFS average-case cost after sorting is $\mathcal{O}(kn)$ (Thm. 8). In this case, only n^f points survived the sorting phase, so LESS's SF phase is bounded above by $\mathcal{O}(kn)$. □

Proving LESS's best-case performance directly is not as straightforward. Of course, it follows directly from the average-case analysis.

Theorem 14 *Under CI (Def. 1) and the model in §2.1, LESS has a best-case runtime of $\mathcal{O}(kn)$.*

Proof 14 *The records have a linear ordering. Thus, this is like considering the average-case runtime for skyline problem with dimensionality one.* □

Worst-case analysis is straightforward.

Theorem 15 *Under CI (Def. 1), LESS has a worst-case runtime of $\mathcal{O}(kn^2)$.*

Proof 15 *Nothing is eliminated in the sort phase, which costs $\mathcal{O}(n\lg n)$. The SF phase costs the same as the worst-case of SFS, $\mathcal{O}(kn^2)$ (Thm. 9).* □

[11] This is simpler to consider than *without* replacement, and is an upper bound with respect to the number of draws needed *without* replacement.

3.4 Issues and Improvements

Since our experiments in §3.2, we have been focusing on how to decrease the CPU load of LESS, and of maximal-vector algorithms generally. LESS and SFS must make $m^2/2$ comparisons just to verify that the maximals are, indeed, maximals. The m^2 is cut in half since the input stream is in sorted order; we know no record can be dominated by any after it. BNL faces this same computational load, and does cumulatively more comparisons as records are compared against non-maximal records in its window.

There are two ways to address the comparison load: reduce further somehow the number of comparisons that must be made; and improve the efficiency of the comparison operation itself. The divide-and-conquer algorithms have a seeming advantage here. DD&C, LD&C, and FLET have a $o(n^2)$ worst-case performance. They need not compare every maximal against every maximal. Of course, §2.4 demonstrates that the divide-and-conquer algorithms have their own limitations.

The sorted order of the input stream need not be the same as that in which the records are kept in the EF and the SF windows. Indeed, we have learned that using two different orderings can be advantageous. (Likewise, this is true for SFS also.) Say that we sort the data in a nested sort with respect to skyline columns, and keep the EF and SF windows sorted by entropy as before. This has the additional benefit that the data can be sorted in a natural way, perhaps useful to other parts of a query plan. Now when a stream record is compared against the SF records, the comparison can be stopped early, as soon as the stream record's entropy is greater than the next SF record's. At this point, we know the stream record is maximal. We have observed this change to reduce the maximal-to-maximal comparisons needed to 26% of that required before for 500,000 points and seven dimensions. This would reduce LESS's performance on the seven dimensional data-set in Fig. 4 from 15 to, say, 7 seconds.

LESS can be used with any tree index that provides a sorted order which is also a topological order with respect to the maximality conditions. In this case, the index replaces the need to sort the data initially. The existence of an applicable index for a given skyline query is much more likely than for the types of index structures the index-based skyline algorithms employ (as discussed in §2.3).[12] The index also can be a standard type commonly employed in databases such as a B+ tree index. It only needs to provide a way to read the data in sorted order.

The dual-order versions of LESS—one order for the input stream and one for the skyline window—that we are investigating have given us insight into how we can handle better sets with anti-correlation. This rep-

[12] To be fair, researchers investigating index-based skyline algorithms are seeking $o(n)$ performance.

resents the worst-case scenario for maximal-vector algorithms (§2.1). We are able to handle well some cases of anti-correlation, for instance, when one dimension is highly anti-correlated with another one. A modified version of LESS will run still in $\mathcal{O}(kn)$ average-case time for this. We believe we shall be able to extend this to handle most anti-correlation effects in the input to still achieve good running time. Furthermore, this may lead to ways to improve on LESS-like algorithms worst-case running time to better than $\mathcal{O}(kn^2)$.

There may be other ways to reduce the computational load of the comparisons themselves. Clearly, there is much to gain by making the comparison operation that the maximal-vector algorithm must do so often more efficient. We are exploring these techniques further, both experimentally and analytically, to see how much improvement we can accomplish. We anticipate improving upon the algorithm significantly more.

4 Conclusions

We have reviewed extensively the existing field of algorithms for maximal vector computation and analyzed their runtime performances. We show that the divide-and-conquer based algorithms are flawed in that the dimensionality k results in very large "multiplicative-constants" over their $\mathcal{O}(n)$ average-case performance. The scan-based skyline algorithms, while seemingly more naïve, are much better behaved in practice. We introduced a new algorithm, LESS, which improves significantly over the existing skyline algorithms, and we prove that its average-case performance is $\mathcal{O}(kn)$. This is linear in the number of data points for fixed dimensionality k, and scales linearly as k is increased.

There remains room for improvement, and there are clear directions for future work. Our proofs for average-case performance rely on a *uniformity* assumption (Def. 1). In truth, LESS and related algorithms are quite robust in the absence of uniformity. We may be able to reduce the number of assumptions that need to be made for the proofs of asymptotic complexity. We want to reduce the comparison load of maximal-to-maximal comparisons necessary in LESS-like algorithms. While the divide-and-conquer algorithms do not work well, their worst-case running times are $o(n^2)$, while LESS's is $\mathcal{O}(n^2)$. It is a question whether the $\mathcal{O}(n^2)$ worst-case of scan-based algorithms can be improved. Even if not, we want an algorithm to avoid worst-case scenarios as much as possible. For maximal vectors, anti-correlation in the data-set causes m to approach n. We want to be able to handle sets with anti-correlation much better. We are presently working on promising ideas for this, as discussed in §3.4.

We have found it fascinating that a problem as seemingly simple as maximal vector computation is, in fact, fairly complex to accomplish well. While there have been a number of efforts to develop good algorithms for finding the maximals, there has not been a clear understanding of the performance issues involved. This work should help to clarify these issues, and lead to better understanding of maximal-vector computation and related problems. Lastly, LESS represents a high-performance algorithm for maximal-vector computation.

References

[1] W.-T. Balke and U. Güntzer. Multi-objective query processing for database systems. In *VLDB*, pp. 936–947, Aug. 2004.

[2] O. Barndorff-Nielsen and M. Sobel. On the distribution of the number of admissible points in a vector random sample. *Theory of Prob. and its Appl.*, 11(2):249–269, 1966.

[3] J.L. Bentley, K.L. Clarkson, and D.B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. In SODA, pp. 179–187, Jan. 1990.

[4] 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. *JACM*, 25(4):536–543, 1978.

[5] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pp. 421–430, 2001.

[6] C. Buchta. On the average number of maxima in a set of vectors. *Information Processing Letters*, 33:63–65, 1989.

[7] J. Chomicki. Querying with intrinsic preferences. In *EDBT*, pp. 34–51, Mar. 2002.

[8] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pp. 717–719, Mar. 2003.

[9] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting: Theory and optimization. In *Int. Inf. Sys. Conference (IIS)* pp. 593–602, Jun. 2005.

[10] P.-K. Eng, B.C. Ooi, and K.-L. Tan. Indexing for progressive skyline computation. *Data and Knowledge Engineering*, 46(2):169–201, 2003.

[11] P. Godfrey. Skyline cardinality for relational processing. In *FoIKS*, pp. 78–97, Feb. 2004.

[12] K.-L. Tan, P.-K. Eng, and B.C. Ooi. Efficient progressive skyline computation. In *VLDB*, pp. 301–310, Sept. 2001.

[13] D. Kossmann, F. Ramask, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pp. 275–286, Aug. 2002.

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

[15] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pp. 467–478, Jun. 2003.