# Algorithms and Analyses for Maximal Vector Computation

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

The maximal vector problem is to identify the maximals over a collection of vectors. This arises in many contexts and, as such, has been well studied. The problem recently gained renewed attention with skyline queries for relational databases and with work to develop skyline algorithms that are external and relationally well behaved.

While many algorithms have been proposed, how they perform has been unclear. We study the performance of, and design choices behind, these algorithms. We prove runtime bounds based on the number of vectors $n$ *and* the dimensionality $k$. Early algorithms based on divide-and-conquer established seemingly good average and worst-case asymptotic runtimes. In fact, the problem can be solved in $\mathcal{O}(n)$ average-case (holding $k$ as fixed). We prove, however, that the performance is quite bad with respect to $k$. We demonstrate that the more recent skyline algorithms are better behaved, and can also achieve $\mathcal{O}(kn)$ average-case. While $k$ matters for these, in practice, its effect vanishes in the asymptotic. We introduce a new external algorithm, LESS, that is more efficient and better behaved. We evaluate LESS's effectiveness and improvement over the field, and prove that 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*. The parameters of the problem are

- $n$, the number of points in the input set;
- $k$, the dimensionality of the space (that is, how many coordinates are used for comparison); and
- $m$, the number of maximal points in the output 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 time it is to find the maximals over a set of *tuples*. Certain columns with ordered domains of the input relation are designated as the *skyline criteria*. Dominance is then defined with respect to these criteria. The non-dominated tuples constitute the *skyline set*.

The idea of skyline queries has attracted much attention since its introduction in [8]. It is thought that skyline offers a good mechanism for incorporating preferences into relational queries. Of course, its implementation would also enable maximal vector applications to be built on relational database systems efficiently. While the idea of maximal vectors 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 efficiently over data sets that are too large to fit in main memory). Other skyline-algorithm work has explored how to use indexes and other preprocessing for more efficient runtime evaluation. This renewed interest in maximal vectors via skyline has offered tangible results in new application areas and in providing relationally well-behaved, external algorithms.

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.

On the one hand, intuition says it should be trivial to design a reasonably good algorithm for finding the maximal vectors. We prove that many obvious approaches have, in fact, $\mathcal{O}(n)$ average-case running time.[1] On the other hand, performance varies widely for the algorithms when applied to input sets of large, but realistic, sizes ($n$) and reasonable dimensionality ($k$). The performance profile can be quite different than as simply suggested by the asymptotic limits.

In truth, designing a good algorithm for the maximal vector problem is far from simple. There are many subtle, but important, issues to address. For some algorithms—namely, the divide-and-conquer approaches—the impact of the dimensionality is profound. For others—namely the skyline algorithms—we prove that they have $\mathcal{O}(kn)$ average-case running time. Even this is deceptive, however, as the "multiplicative constant" becomes reasonable only in the extreme limit of $n$. Design choices that appear innocent can lead to quite different performance profiles.

This paper is a direct extension of the work in [20]. In this paper, as in [20], we focus on *generic* maximal-vector algorithms; that is, on algorithms for which expensive preprocessing steps or data-structures such as indexes are not required.

In §2, we discuss the maximal vector problem. We illustrate with an example (§2.1), discuss a general approach to finding the maximal vectors (§2.2), and outline criteria for a good maximal-vector algorithm (§2.3). We then introduce a *base set of assumptions* on which we can base average-case analyses (§2.4). This employs an estimate of the number of maximals expected ($\widehat{m}$), on average, assuming *statistical independence* of the dimensions and *sparseness* (distinct values) of the vectors along each dimension. In some cases for analysis, a third assumption of *uniformity*—that the values along a dimension are uniformly distributed—is needed.

In §3, we simultaneously review the work in the area and analyze the proposed algorithms' runtime performances. We summarize the generic algorithmic approaches—both older algorithms and newer, external skyline algorithms—for computing maximal vector sets (§3.1). We formally analyze the runtime performances of the existing generic algorithms, especially with consideration of the dimensionality $k$'s impact, to identify the bottlenecks and compare advantages and disadvantages between the approaches. We address the divide-and-conquer approaches in §3.2, and then the external, skyline approaches in §3.3. This recapitulate the results from [20] and addresses previously unresolved issues, such as the average runtime performance of the algorithm BNL from [8].[2] We extend our coverage of related algorithms. (We additionally address the skyline algorithm Best [17, 33].) We discuss the design choices behind the skyline algorithms, and the ramifications of these on performance.

[1]Meanwhile, that their average-case runtimes are $\mathcal{O}(n)$ is anything but obvious.

[2]Some of the new results in this paper (not in [20]) were presented in our talk for [20] in Trondheim at VLDB 2005.

In §3.4, we present a new algorithm for maximal vector computation, LESS (*linear elimination sort for skyline*), that essentially combines aspects of a number of the established algorithms, and offers a substantial improvement over the field. The design of LESS is motivated by our earlier analyses and observations (§3.4.1). We present an experimental evaluation of LESS that demonstrates its improvement over the existing field (§3.4.2). 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 (§3.4.3).

In §3.5, we identify the key bottlenecks for any maximal-vector algorithm, and discuss ways that LESS—and other skyline algorithms—could be further improved In §3.6, discuss how the the assumptions behind the average-case analyses (§2.4) can be relaxed, and how these affect the algorithms. This strengthens the results from [20] by showing the extent to which the assumptions can be lifted. (We prove that FLET, SFS, and LESS have $\mathcal{O}(kn)$ average-case runtime *without* the uniformity assumption.)

In §4, we discuss other work related to maximal vectors and skyline that were not discussed earlier. We briefly discuss some of the index-based approaches to maximal vector computation, and why index-based approaches are necessarily of limited utility.

In §5, we discuss future issues and conclude.

## 2 The Maximal Vector Problem

### 2.1 An Example

Consider a table for hotels with columns name, address, dist (distance to the beach), stars (quality rating), and price, as data in Figure 1. This table has three metric columns dist, stars, and price. Based on these three metric dimensions, we could visualize the data as points in three dimensional space.

In [8], they introduced a hypothetical extension to the SQL query language called *skyline* to allow one to query for maximals. The skyline query in Figure 2 over the hotel table asks for hotels with the most stars, that are closest to the beach, and are the least expensive.

| name | stars | dist | price |
|---|---|---|---|
| Aga | ⋆⋆ | 0.7 | 1,175 |
| Fol | ⋆ | 1.2 | 1,237 |
| Kaz | ⋆ | 0.2 | 750 |
| Neo | ⋆⋆⋆ | 0.2 | 2,250 |
| Tor | ⋆⋆⋆ | 0.5 | 2,550 |
| Uma | ⋆⋆ | 0.5 | 980 |

Figure 1: The hotel table.

The semantics of a skyline query is to find the maximals, throwing away any tuples that are dominated by others. The rows in black in the table (Figure 1) are the answers to the query; the rows that are grayed out are those that were dominated. Aga can be eliminated, for example, by comparison with Uma. Fol is eliminated by comparison with Aga, Kaz, or Uma. Tor is eliminated by comparison with Neo. None of Kaz, Neo, or Uma is dominated by any other, however, so these are the answers.

```
select name, address
    from Hotel
    skyline of stars max,
            dist min,
            price min
```

Figure 2: The hotel query.

There may be many maximals. A maximal need not be best on any one criterion. For example, Uma does not have the most stars, is not the closest to the beach, and is not the least expensive. Rather, it represents a good balance of the criteria.

## 2.2 An Algorithm

To find the *maximum value* from an unordered list of values can be, of course, accomplished in a single pass over the list with $n-1$ comparisons. If it is a list of *records* ("vectors") instead and one is to find each record that has the maximum value with respect to a given field (say, *val*), then the problem is complicated only slightly. Records may share the same value on *val*, so there may be *ties* for the maximum. In this case, it would suffice to sweep the list twice: the first time to find the maximum *val*; and the second time to collect the records with that *val*. This is, of course, just the maximum-vector problem with one dimension ($k = 1$).

For the multi-dimensional maximum-vector problem ($k > 1$), should it be much harder? Let $\mathcal{L}_n$ ($\vec{t}_0, \ldots, \vec{t}_{n-1}$) be the list of vectors. Let $\vec{t}_i \succ \vec{t}_j$ denote that $\vec{t}_i$ ties or is higher than $\vec{t}_j$ on each of the $k$ components, and it is strictly higher on at least one; that is, that $\vec{t}_i$ *dominates* $\vec{t}_j$.

Essentially, the same basic strategy can be made to work. A paraphrase of the Best algorithm from [17, 33] (and discussed in [32]) is shown in Figure 3. In a pass over the list, one can find *a* maximal. In subsequent passes over the list, additional maximals are found. On each pass, any vector found to be dominated by the current maximal-so-far ($\vec{b}$) is eliminated from the list. The second *foreach* loop is necessary after the final maximal for the pass was found ($\vec{b}$) to clean up by removing any vector dominated by the final maximal. (One need only check through the list *up to* the point when the final maximal was stored in $\vec{b}$.)

```
while ($\mathcal{L}$ is not empty)
    $\vec{b}$ = shift($\mathcal{L}$)      // Get the first.
    foreach $\vec{t}$ in $\mathcal{L}$    // Find a max.
        if ($\vec{b} \succ \vec{t}$)
            remove $\vec{t}$ from $\mathcal{L}$
        else if ($\vec{t} \succ \vec{b}$)
            remove $\vec{t}$ from $\mathcal{L}$
            $\vec{b}$ := $\vec{t}$
    report $\vec{b}$
    foreach $\vec{t}$ in $\mathcal{L}$    // Clean up.
        if ($\vec{b}$.rank $<$ $\vec{t}$.rank and $\vec{b} \succ \vec{t}$)
            remove $\vec{t}$ from $\mathcal{L}$
        else if ($\vec{b}$.rank $>$ $\vec{t}$.rank)
            break
```

Figure 3: The Best algorithm.

Given $m$ maximals in the list, we know the *while* loop will iterate exactly $m$ times, finding a new maximal each time. After $m$ iterations, the list $\mathcal{L}$ will be empty.

One way to define best, average, and worst-case is based on how many maximals ($m$) there are. Thus, in best-case, there is just a single maximal (so one *maximum*). Best will make at most two full passes of $\mathcal{L}$ and make $\mathcal{O}(n)$ number of vector comparisons. Each vector comparison involves $k$ unit comparisons to compare the components, so we say this is $\mathcal{O}(kn)$ amount of work.

In worst-case, every vector is a maximal ($m = n$). No vectors are eliminated from $\mathcal{L}$; one is removed as a maximal in each pass. Therefore, the worst-case running time for Best is $\mathcal{O}(kn^2)$.

We know then a ceiling on the average-case running time for Best is $\mathcal{O}(kmn)$, using "$m$" in the bound. We could be more specific if we knew what the value $m$ is—that is, how many maximals are to be expected—on average. (We do know this, and this is discussed in §2.4.) Furthermore, this is quite likely a loose ceiling since many vectors may be eliminated from $\mathcal{L}$ on each pass. A floor on Best's running time is $\mathcal{O}(km^2)$, since, in the least, each maximal is compared against every other maximal.

To determine how good then Best is—and how good the other maximal-vector algorithms are, as well—in average-case, we need to define a reasonable model under which to measure best, average, and worst-case. We do this in §2.4. In the next section, we consider other desired criteria, besides just runtime performance, by which the algorithms might be judged.

## 2.3 Criteria for a Good Algorithm

Of course, runtime performance is of primary importance. A runtime of $\mathcal{O}(n^2)$ is untenable for large data sets. In database systems, $\mathcal{O}(n \lg n)$ is considered an expensive operation, such as external sort. Some maximal-vector algorithms have been shown to have $\mathcal{O}(n)$, linear, average-case performance. This should be our target.[3]

We also want an *external* algorithm that can handle input sets that are too large for main memory. An external algorithm must be *I/O conscious.* That is, the algorithm should be well behaved in how many I/O's are spent, in addition to the CPU (computational) load. The latter is measured, in part, by asymptotic runtime analyses. In this work, we focus on the computational load. We do not model formally I/O expenditure. However, we do pay careful attention to which strategies are amenable to good I/O performance.

There are additional criteria by which we might judge maximal-vector algorithms. In [28], a useful categorization of existing skyline algorithms is presented (which follows on the work in [21, 24]). They use the criteria enumerated below to characterize the behavior and applicability of the algorithms. These are properties that they and we would like a maximal-vector algorithm to have.

1. *progressiveness.* The first results should be returned almost instantly, and the output size should gradually increase.
2. *absence of false hits.* The output should contain only the skyline points (maximals).
3. *absence of temporary false hits.* The algorithm should not discover temporary "skyline" points ("maximals") that will be later discarded.
4. *fairness.* The algorithm should not favor points that are particularly good on one dimension (but not necessarily others).
5. *incorporation of preference.* It should be possible to specify an order by which the skyline points (maximals) are reported.
6. *universality.* The algorithm should be applicable to any data-set distribution and dimensionality.

Likewise, we can employ these as design criteria.

### 2.4 Assumptions for Analysis

Performance of maximal-vector algorithms depends on the number of maximals ($m$). We shall consider average-case performance based on the expected value of the number of maximals ($\widehat{m}$). To establish an expected value of $m$, we shall need to make certain assumptions about the input set. First, let us consider when a set of points is *normalized.*

**Definition 1** *A set of points is* normal *if the values of the points along any given dimension fall uniformly along the open interval* $(0, 1)$. *(This is visualized in Figure 4 for* $k = 3$.*)*

*If a set of points is not normal, it can be* normalized *into a normal set that is* rank-equivalent *(and so has the same maximals). A procedure for this would be as follows. For each dimension* $d_i$*, find the* ordinal rank*—as by sorting—of each point,* $0, \ldots, V_i - 1$. $V_i$ *represents the number of distinct values on dimension* $i$ $(d_i)$ *over the points. Rank* $0$ *is assigned for the lowest value on* $d_i$*, and so forth. For each point in the normalized set, given its rank on* $d_i$ *is* $j$*, assign it the value* $(j + 1)/(V_i + 1)$.

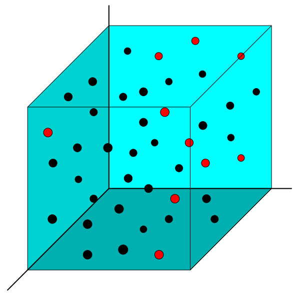


Figure 4: A normalized 3-$d$ set.

*We shall refer to the* normalized set of points *as the set of points transformed in this way.*

**Definition 2** *Consider the following properties of a set of points.*

*a.* independence. *The values of the* normalized *set of points over a single dimension are statistically independent of the values along any other dimension.*

*b.* sparseness (distinct values). *Points (mostly) have distinct values along any dimension (that is, there are not many repeated values).*

*c.* uniformity. *The values of the points along any one dimension are uniformly distributed.*

*Collectively, the properties of* independence *and* sparseness *are called* component independence *(CI) [6]. Let us call collectively the three properties* uniform independence *(UI).*

*Note that for a set of points that has CI, the normalized set of points has UI. Additionally, assume under* uniformity *that any value is on the interval* $(0, 1)$.[4] *Thus, UI implies a normal set.*

It is important to note that *independence* is defined with respect to the *normalized* set. *Rank correlation* is the measure of linear correlation between two lists ("dimensions") of values that have been replaced by their ordinal ranks (as in the normalized set). Thus, independence states that each pair of dimensions has a rank correlation of zero. It is interesting to note that an un-normalized set may have apparent non-zero linear correlations even when all its pair-wise rank correlations are zero (and thus be independent by our definition).

[3]Sub-linear performance would be even better. However, this is not possible for a *generic* algorithm that requires no extensive pre-processing or data-structures. Therefore, in the least, every vector will need to be scanned.

[4]Uniformity itself does not assure the values on a given dimension are on $(0, 1)$. However, mapping the values onto $(0, 1)$ is inexpensive. Knowing the maximum and minimum values of the points for each dimension is sufficient to make this mapping.

In many cases, we only need to assume CI, as the property of *uniformity* will not be necessary for the result. In other cases, we need further to assume that the set is normalized (thus additionally assume uniformity). We must note that if the data must be actually transformed into a normalized set, this transformation is not computationally insignificant.

Under CI, the expected value of the number of maximals—denote this by $\widehat{m}$—is known [9, 19]: $\widehat{m} = \mathsf{H}_{k-1,n}$, 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 $k > 1$. $\mathsf{H}_{k,n} \approx \mathsf{H}_{1,n}^{k}/k! \approx ((\ln n) + \gamma)^k/k!$.

For best-case, assume that there is a total ordering of the points, $\vec{t}_0, \ldots, \vec{t}_{n-1}$, such that any $\vec{t}_i$ dominates all $\vec{t}_j$, for $i < j$. Thus, in best-case, $m = 1$ (the one point being $\vec{t}_0$).[5]

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 $k$.

## 3 Algorithms and Analyses

In §3.1, we make our first pass over the algorithms, describe the algorithms, show their best and worst-case runtimes, and discuss known average cases. In §3.2, we return to the divide-and-conquer algorithms to prove their average-case runtimes with $k$ considered. In §3.3, we do likewise for the skyline algorithms. In §3.4, we introduce a new algorithm, LESS, that improves on the previous. In §3.5, we consider issues and optimization opportunities for LESS and the skyline algorithms. In §3.6, we discuss the extent to which the assumptions from §2.4 can be relaxed.

### 3.1 The Algorithms

The main (generic) algorithms that have been proposed for maximal vectors are listed in Figure 5. 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 runtime analyses—with respect to CI or UI and our model for average case from §2.4—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. The rest are proved in the indicated theorems.[6]

The first group consists of divide-and-conquer-based algorithms. DD&C (*double divide and conquer*) [25], LD&C (*linear divide and conquer*) [6], and FLET (*fast linear expected time*) [5] are "theoretical" algorithms that were proposed to establish the best bounds possible on the maximal-vector problem. No attention was paid in this work 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 $d_{k-1}$, with respect to the the sorted order over $d_{k-1}$. 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 $d_{k-1}$ than those in $\mathcal{B}$ (given the original set was divided over the sorted list of points with respect to dimension $d_{k-1}$). The maximals of $\mathcal{A} \cup \mathcal{B}$ are determined by applying DD&C, but now over dimensions $d_0, \ldots, d_{k-2}$, so with reduced dimensionality.[7]

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. Thus, DD&C establishes that the maximal-vector problem is, in fact, $o(n^2)$. 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 §3.2.

LD&C [6] 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

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

Note that $(\ln^{k-1} n) \lg^{k-2}(\ln^{k-1} n)$ is $o(n)$. Therefore,

[5] We consider a total order so that, for any subset of the points, there is just one maximal with respect to that subset. This is necessary for discussing the divide-and-conquer-based algorithms.

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

[7] All points in $\mathcal{A}$ are marked so none will be thrown away. Note that only points in $\mathcal{B}$ can be dominated by points in $\mathcal{A}$, since those in $\mathcal{A}$ are better along dimension $d_{k-1}$.

| algorithm | | ext. | best-case | | average-case | | worst-case | |
|---|---|---|---|---|---|---|---|---|
| DD&C | [25] | no | $\mathcal{O}(kn\lg n)$ | §3.1 | $\Omega(kn\lg n + (k-1)^{k-3}n)$ | Thm.11 | $\mathcal{O}(n\lg^{k-2} n)$ | [25] |
| LD&C | [6] | no | $\mathcal{O}(kn)$ | §3.1 | $\mathcal{O}(n)$, $\Omega((k-1)^{k-2}n)$ | [6], Thm. 10 | $\mathcal{O}(n\lg^{k-1} n)$ | [6] |
| FLET | [5] | no | $\mathcal{O}(kn)$ | §3.1 | $\mathcal{O}(kn)$ | [5] | $\mathcal{O}(n\lg^{k-2} n)$ | [5] |
| SD&C | [8] | – | $\mathcal{O}(kn)$ | Thm. 3 | $\Omega(\sqrt{k}2^{2k}n)$ | Thm. 9 | $\mathcal{O}(kn^2)$ | Thm. 4 |
| BNL | [8] | yes | $\mathcal{O}(kn)$ | Thm. 5 | $\mathcal{O}(kn)$ | Thm. 12 | $\mathcal{O}(kn^2)$ | Thm. 6 |
| Best | [17, 33] | no | $\mathcal{O}(kn)$ | §2.2 | $\mathcal{O}(kn)$ | Thm. 15 | $\mathcal{O}(kn^2)$ | §2.2 |
| SFS | [15, 16] | yes | $\mathcal{O}(n\lg n + kn)$ | Thm. 7 | $\mathcal{O}(n\lg n + kn)$ | Thm. 17, 19 | $\mathcal{O}(kn^2)$ | Thm. 8 |
| LESS | – | yes | $\mathcal{O}(kn)$ | Thm. 21 | $\mathcal{O}(kn)$ | Thm. 20 | $\mathcal{O}(kn^2)$ | Thm. 22 |

Figure 5: The generic maximal vector algorithms.

LD&C is average-case linear, $\mathcal{O}(n)$ [6].

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 resulting 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 §3.2, 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 [5] takes a rather different approach to improving on DD&C's average-case. Under UI,[8] 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 [6] and [25], 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 [8], along with two general algorithms proposed for computing the skyline in the context of a relational query engine.

The first general algorithm in [8] 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 $\mathcal{A} \cup \mathcal{B}$.

**Theorem 3** *Under CI (Def. 2) and the model in §2.4,* SD&C *has a best-case runtime of* $\mathcal{O}(kn)$. [20]

**Proof 3** *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 §3.2.)

**Theorem 4** *Under CI (Def. 2),* SD&C *has a worst-case runtime of* $\mathcal{O}(kn^2)$. [20]

**Proof 4** *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

[8]For the analysis of FLET, we need to make the additional assumption of *uniformity* from Def. 2.

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 [8] is BNL, *block nested loops*. This is essentially a generalization of the intuitive approach, Best, discussed in §2.2, and works remarkably well.[9] 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. Best is essentially BNL then with a window size of one tuple.

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. Otherwise, dominated window points are still eliminated as before. If the new point is not dominated *and* no space was freed in the window, it is written to an overflow file. The creation of an overflow file means that 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.[10] 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 "maximals"—maximal with respect to the list seen so far—are much more effective at eliminating other points than are the local maximals computed at each recursive stage in divide-and-conquer.

**Theorem 5** *Under CI (Def. 2) and the model in §2.4,* BNL *has a best-case runtime of* $\mathcal{O}(kn)$. [20]

**Proof 5** 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.* □

Let $w$ be the size limit of the window in number of points.

[9]However, BNL [8] precedes Best [17, 33] in the literature.

[10]This can be done as we did in the Best algorithm in Figure 3 by assigning initially each point (tuple) a *rank*, its initial position in the list.

**Theorem 6** *Under CI (Def. 2),* BNL *has a worst-case runtime of* $\mathcal{O}(kn^2)$. [20]

**Proof 6** *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 [15], 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 $d_0, \ldots, d_{k-1}$, for instance, would suffice. In [16], 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 [15, 16], sorting the records by *volume* descending, $\prod_{i=1}^{k} \vec{t}\,[i]$;[11] (or, equivalently, by *entropy* descending, $\sum_{i=1}^{k} \ln \vec{t}\,[i]$,[12] with the guarantee that the values $\vec{t}\,[i] > 0$ for all records $t$ and dimensions $i$) is advocated. This has the advantage of tending to push records that dominate many records towards the beginning of the stream.

Under UI, the number of records a given record dominates is proportional to its volume. Thus, by placing records with higher volumes 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 §3.4 and in the proof that LESS is $\mathcal{O}(kn)$ (Thm. 20).

SFS maintains a main-memory window as does BNL. However, in SFS, it is impossible for a point off the stream to dominate any of the points already in the window. Any point is 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 emptied 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. Therefore, 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 win-

[11]In this case, as in the discussion about FLET, we are assuming a normalized set (Def. 1, and so, additionally, the *uniformity* assumption (Def. 2.

[12]Keeping entropy instead of volume helps to avoid register overflow or underflow.

dow, it can also be shipped as a maximal to the next operation.

**Theorem 7** *Under CI (Def. 2) and the model in §2.4,* SFS *has a best-case runtime of* $\mathcal{O}(kn + n\lg n)$. [20]

**Proof 7** *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.* □

**Theorem 8** *Under CI (Def. 2),* SFS *has a worst-case runtime of* $\mathcal{O}(kn^2)$. [20]

**Proof 8** *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.* □

In the experiments in [15], SFS—*including* SFS's necessary sorting step—performed better I/O-wise, and ran in better time, than BNL. The experiments in [15] were run over million-tuple data sets and with dimensions of five to seven. In truth, however, we and others have found that it is quite difficult to compare SFS, BNL, and other maximal vector algorithms reliably, either experimentally or analytically. Their performance depends greatly on the implementation details of the algorithms, on the size of the input (the number of records, $n$, and the dimensionality, $k$), on the nature of the data-set (data distributions, stream order, and so forth), and on runtime parameters (for instance, buffer pool allocation).

In this paper, BNL fares much better experimentally (shown in §3.4.2), as the data-sets used were larger and, we believe, we have a more efficient implementation. In this case, the sort step dominates SFS's cost. In §3.3, however, we demonstrate analytically that SFS algorithm has significant advantages over BNL in reducing the number of comparisons needed. So, on the one hand, SFS may make much fewer comparisons than BNL, since SFS compares only against maximals but BNL often compares against non-maximals. On the other hand, SFS *does* require sorting, and this cost can dominate its performance.

## 3.2 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 runtimes, 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 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, however.

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 [8] as a viable external divide-and-conquer for computing maximal vectors. As we argued above, and as is argued in [27], 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 9** *Under CI (Def. 2),* SD&C *has average-case runtime of* $\Omega(\sqrt{k}2^{2k}n)$. [20]

**Proof 9** *Let* $n = 2^q$ *for some positive integer* $q$*, without loss of generality. Consider the function* $T$ *as follows.*

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

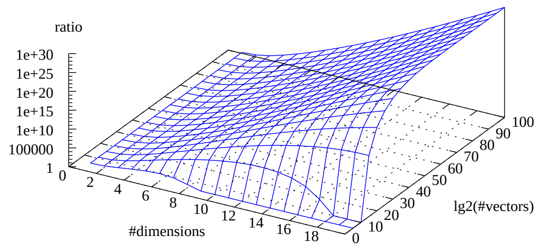


Figure 6: Behavior of LD&C.

*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 [4], 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.* □

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 [8], 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 [8]. 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. 9 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

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

We plot this in Figure 6.[13] 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 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 10** *Under CI (Def. 2),* LD&C *has average-case runtime of* $\Omega((k-1)^{k-2}n)$. [20]

**Proof 10** *Let $n = 2^q$ for some positive integer $q$, without loss of generality.*

$$\begin{aligned}
&m_{k,n}\lg_2^{k-2} m_{k,n}\\
&\quad\approx ((\ln^{k-1} n)/(k-1)!)\lg_2(((\ln^{k-1} n)/(k-1)!))^{k-2}\\
&\qquad c_1 = 1/(k-1)!\\
&\quad= c_1(\ln 2)^{k-1}(\lg_2 n)^{k-1}\\
&\qquad \cdot (\lg_2((\ln 2)^{k-1}(\lg_2 n)^{k-1}) - \lg_2(k-1)!)^{k-2}\\
&\qquad c_2 = \ln^{k-1} 2\\
&\quad> c_1c_2(\lg_2^{k-1} n)\\
&\qquad \cdot ((k-1)((\lg_2 q) + (\ln 2) - (\lg_2(k-1)))^{k-2}\\
&\qquad \textit{when } \lg_2 q > \lg_2(k-1)\\
&\quad> c_1c_2(\lg_2^{k-1} n)(k-1)^{k-2}\\
&\qquad \textit{when } \lg_2 q - \lg_2(k-1) \geq 1\\
&\qquad \textit{thus } q \geq 2(k-1)
\end{aligned}$$

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

$$\begin{aligned}
&T(n) = 2T(n/2) + \mathsf{max}(m_{k,n}\lg_2^{k-2} m_{k,n}, 1)\\
&\quad> c_1c_2(k-1)^{k-2}\sum_{i=l}^{q} 2^{q-i} i^{k-1}\\
&\qquad \textit{for } n \geq 2^l\\
&\quad= c_1c_2(k-1)^{k-2}2^l \sum_{i=0}^{(q-l)} 2^{(q-l)-i} i^{k-1}\\
&\quad\approx c_1c_2(k-1)^{k-2}2^l(\lg_2^{k-1} e)(k-1)!2^{(q-l)}\\
&\quad= (k-1)^{k-2}2^q\\
&\quad= (k-1)^{k-2}n
\end{aligned}$$

*$T(n)$ 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 11** *Under CI (Def. 2),* DD&C *has an average-case runtime of* $\Omega(kn\lg n + (k-1)^{k-3}n)$. [20]

**Proof 10** DD&C *first does a divide and conquer over the data on the first dimension. During a merge step*

[13] 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.

*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.*

$T(n) = 1$

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

*By the same proof steps as in the proof for Thm. 10, 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 runtimes 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.

## 3.3 The Skyline Algorithms

In some respects, the skyline algorithms are simpler than previous maximal-vector algorithms. They are scan-based and so are more amenable to externalizing. Their worst-case runtimes are worse than most of the divide-and-conquer algorithms; however, they are no worse in any practical sense. (For example, DD&C's worst-case of $kn\lg n + (k-1)^{k-3}n$ steps *is* asymptotically better than, say, BNL's $kn^2$, but these two functions only cross for a reasonable $k$ at extreme $n$.) How do the skyline algorithms perform in average case, though? Surprisingly, we have learned that the skyline algorithms are *significantly* better than the previous algorithms. We prove that here.

In other respects, the skyline algorithms are more complex than their earlier counterparts. There are more design choices involved, and these choices can have dramatic effects on runtime performance. Furthermore, it has been—and, in many cases, remains—quite opaque as to why these lead to the behaviors they do.

For BNL and SFS (and later, LESS), the following policies must be determined.

- *window management policy*: How should window points (tuples) be organized in the window? In what order are they to be compared against a candidate from the stream?
- *window size* ($w$): This is likely a runtime parameter that a query processor would set. How does window size affect the runtime performance?

For SFS (and later, LESS), there is an additional policy to decide.

- *stream order*: How should the stream be sorted prior to processing? How does the choice of stream order affect performance?

To consider BNL's average-case runtime, let us simplify these parameters. For now, we consider that the window size is effectively unbounded; that is, it is sufficiently big ($w \gg m$) that a second pass is not required. For BNL's window management policy, when a tuple is added to the window, it is appended at the end of the window list. When a candidate tuple is compared against the window, it is compared one-by-one against the window list in order. Call this window policy *append*.

Let the input *stream*—or likewise called *list* or *table*—$\mathcal{L}_n$ consist of $\vec{t}_0, \ldots \vec{t}_{n-1}$ so ordered. Define the partial list $\mathcal{L}_i$ as $\vec{t}_0, \ldots \vec{t}_{i-1}$. Define $\nabla_i$, the *intermediate skyline set* $i$, as the the set of maximals with respect to $\mathcal{L}_i$. Note that not all tuples in $\nabla_i$ are maximals with respect to $\mathcal{L}_n$; some tuple $\vec{t}_j$ for $j \geq i$ may dominate some of the tuples in $\nabla_i$. In BNL, each $\vec{t}_i$ is considered, in turn. Let us call the time at which $\vec{t}_i$ is considered *stage* $i$.

At the beginning of stage $i$, the window must hold exactly the tuples from $\nabla_i$.[14] By the append window policy, these will be ordered in the window in the same order that they appeared in the stream. Thus, $\vec{t}_i$ is compared against the tuples in $\nabla_i$. The expected value of the size of $\nabla_i$ is $\widehat{m}_{k,i} = \mathsf{H}_{k-1,i}$. The number of $\nabla_i$ tuples that BNL will compare $\vec{t}_i$ against depends upon how many of the $\nabla_i$ tuples dominate $\vec{t}_i$. If all do, then the first comparison suffices to eliminate $\vec{t}_i$. If none does, $\vec{t}_i$ will be compared against all the $\nabla_i$ tuples (and then itself be appended to the window, and be thus in $\nabla_{i+1}$). If only one $\nabla_i$ tuple dominates $\vec{t}_i$, during the walk through the window, BNL would compare $\vec{t}_i$ against $(|\nabla_i| + 1)/2$ tuples, on average. (The window is randomly ordered because the stream is randomly ordered.) And so forth.

How many of the (local) maximals from $\nabla_i$ dominate $\vec{t}_i$ depends on $\vec{t}_i$'s coordinates. Let function $\mathbf{mttf}_k$—for *mean time to failure*—measure how many comparisons are made for some $\vec{t}_i$ against $\nabla_i$. We want to determine the expected value of this, $\widehat{\mathbf{mttf}}_k$. The function $\mathbf{mttf}_k$ takes as arguments the $k$ coordinates of a tuple (e.g., $\vec{t}_i$) and an argument $j$ for how many points have been previously seen (e.g., $i$).

For the argument's sake, assume the point set $\mathcal{L}_n$ is normalized. Under CI, $\vec{t}_i$ has a uniform probability of falling anywhere in $(0,1)^k$, the $k$-cube. Therefore, the expected value of the number of comparisons made at

[14] For the first stage, let $\nabla_0 = \{\}$.

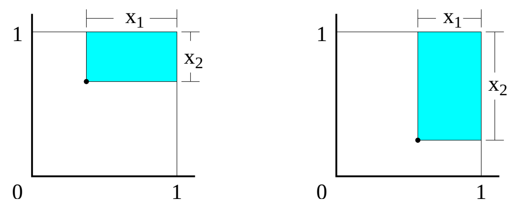


Figure 7: The dominating box of $\vec{t_i}$.

stage $i$ is

$$\int_{x_{k-1}=0}^{1} \ldots \int_{x_0=0}^{1} \widehat{\mathbf{mttf}}_k(x_0, \ldots, x_{k-1}, i) dx_0 \ldots dx_{k-1}$$

We can estimate a ceiling on $\widehat{\mathbf{mttf}}_k(x_0, \ldots, x_{k-1}, i)$. Given $\vec{t_i}$'s coordinates as $x_0, \ldots, x_{k-1}$, we know that of the preceding $i$ points, $(1 - x_0) \cdot \ldots \cdot (1 - x_{k-1}) \cdot i$ of them are expected to fall into the "box" (the $k$-sub-cube) of space that dominates $\vec{t_i}$. (This is visualized in Figure 7 for $k = 2$.) The maximals of these points will be found in $\nabla_i$, of course, but $\nabla_i$ contains additionally other maximals that do not dominate $\vec{t_i}$. By CI, we expect there to be $\mathsf{H}_{k-1,(1-x_0)\cdot\ldots\cdot(1-x_{k-1})\cdot(i)}$ of these. The ratio $\mathsf{H}_{k-1,(1-x_0)\cdot\ldots\cdot(1-x_{k-1})\cdot i}/\mathsf{H}_{k-1,i}$ then is the expected portion of maximals in $\nabla_i$ that dominate $\vec{t_i}$ (of known coordinates).

The inverse ratio then

$$\frac{\mathsf{H}_{k-1,i}}{\mathsf{H}_{k-1,(1-x_0)\cdot\ldots\cdot(1-x_{k-1})\cdot i}}$$

is the expected value of how many $\nabla_i$ BNL would compare $\vec{t_i}$ against, if it selected tuples randomly from $\nabla_i$ *with replacement* (and, in this case, with the proviso that *some* $\nabla_i$ tuple dominates $\vec{t_i}$). This then is a strict upper-bound on the expected value *without replacement*. BNL under the append window policy will walk over the window tuples in order. However, since the (local) maximals in $\nabla_i$ are randomly ordered, this is the same as random selection without replacement.

We can approximate H by ln. Let us define lh though as

$$\mathrm{lh}\, x = \begin{cases} \gamma & \text{if } x < 1 \\ (\ln x) + \gamma & \text{otherwise} \end{cases}$$

This is better behaved for $0 \leq x < 1$, and adds in $\gamma$ to approximate more closely H. We now can approximate $\mathsf{H}_{k-1,i}/\mathsf{H}_{k-1,(1-x_0)\cdot\ldots\cdot(1-x_{k-1})\cdot i}$ by the strict ceiling $\mathrm{lh}^{k-1} i/\mathrm{lh}^{k-1}(1 - x_0) \cdot \ldots \cdot (1 - x_{k-1}) \cdot i$. A ceiling on the expected value of the number of comparisons for $\vec{t_i}$ of unknown coordinates is thus

$$\int_{x_{k-1}=0}^{1} \ldots \int_{x_0=0}^{1} \frac{\mathrm{lh}^{k-1} i}{\mathrm{lh}^{k-1} x_0 \ldots x_{k-1} i} dx_0 \ldots dx_{k-1}$$

Summing this for each stage provides a bound on the average number of comparisons made by BNL:

$$\sum_{i=1}^{n} \int_{x_{k-1}=0}^{1} \ldots \int_{x_0=0}^{1} \frac{\mathrm{lh}^{k-1} i}{\mathrm{lh}^{k-1} x_0 \ldots x_{k-1} i} dx_0 \ldots dx_{k-1}$$

Thus, a ceiling on the number of comparisons made *per vector*, on average, is

$$\int_{z=0}^{1} \int_{x_{k-1}=0}^{1} \ldots \int_{x_0=0}^{1} \frac{\mathrm{lh}^{k-1} zn}{\mathrm{lh}^{k-1} x_0 \ldots x_{k-1} zn} dx_0 \ldots dx_{k-1} dz$$

**Theorem 12** *Under CI,* BNL *with an unbounded window and a window policy of append, in average-case, in the limit of n, makes one comparison per input vector. This is an average-case runtime of $\mathcal{O}(kn)$.*

**Proof 12** *It is easy to show that*

$$\lim_{n\to\infty} \int_{z=0}^{1} \int_{x_{k-1}=0}^{1} \ldots \int_{x_0=0}^{1} \frac{\mathrm{lh}^{k-1} zn}{\mathrm{lh}^{k-1} x_0 \ldots x_{k-1} zn} dx_0 \ldots dx_{k-1} dz = 1 \qquad \square$$

This is a remarkable result. No maximal-vector algorithm could do *better* than one comparison per vector! It may seem, at this point, there is no possibility to improve on this. However, there is. BNL does not converge *quickly* on this ideal, so this asymptotic result is somewhat misleading. The dimensionality $k$ does play a role, as it did with the divide-and-conquer algorithms, even though, in this case, any effect of $k$ asymptotically vanishes in the extreme limit. For data sets of realistic size and dimensionality, BNL has an appreciable "multiplicative constant", which we shall discuss.

First though, we made assumptions for the above analysis of BNL: *append* as the window management policy; an *unbounded* window size; and that the input stream is *randomly ordered*. Since *append* works here, we could certainly choose it to be BNL's window policy. However, BNL would not be a useful external algorithm if its main-memory requirement is dictated by the size of its input. Is BNL still efficient if the window size is not unbounded? Is BNL still efficient if the input stream is ordered?

Let us consider BNL at the opposite extreme with a window size of one ($w = 1$). This is effectively the Best algorithm.[15] First, we must recognize that Best will not find the maximals in a random order, even though the input list is randomly ordered.

[15]There is a small difference between Best as described in Figure 3 and BNL with $w = 1$. BNL, once a maximal has gone full cycle, will add the *next* tuple from the stream into the empty window. Best, however, would commence at the beginning of the stream (list) again.

This difference is unimportant, however, for analysis since we assume that the stream is randomly ordered. The same tuples are eliminated by both algorithms in a pass, so where in the stream the next pass commences is immaterial for the average-case analysis.

**Lemma 13** *Under CI, in each pass, it is more probable that* Best *finds a maximal with a higher volume than one with a lower volume.*

**Proof 13** *It can be shown that* Best*'s procedure will arrive at a maximal with higher volume with higher probability. (Rather, the chance that* Best*'s window slot accrues a vector* outside *a smaller volumed maximal is greater, thus ruling it out as the ultimate selection.)* □

**Lemma 14** *A vector drawn randomly from a normalized set of vectors is eliminated in fewer comparisons against a list of the maximals that is correlated by volume descending—but not more strongly correlated with the volume for any subset of $r$ dimensions than with any other subset of $r$ dimensions, for $r < k$—on average, than against a list of the maximals randomly ordered.*

**Proof 14** *This can be shown to follow since the probability that the vector is eliminated in comparison to a given maximal is proportional to the maximal's volume.* □

Now we prove that Best has average-case runtime of $\mathcal{O}(kn)$ under CI.

**Theorem 15** *Under CI,* BNL *with a window size of one, and hence,* Best*, have an average-case runtime of $\mathcal{O}(kn)$.*

**Proof 15** *We prove this by induction.*

property. Best—BNL *with a window size of one—makes the same number or fewer comparisons than* BNL-open—BNL *with an unbounded window with the append window policy—makes, on average.*

base case. *Consider an input of size one.*

hypothesis. *For some $r > 1$, assume the property for lists of vectors randomly ordered under CI of length $r$ or fewer.*

induction. *Consider a randomly ordered list of size $r + 1$ under CI, $\vec{t}_0, \ldots, \vec{t}_r$. Consider $\vec{t}_r$. $\nabla_r$ contains all the maximals, excepting $\vec{t}_r$ if it is a maximal.*

BNL-*open will compare against the maximals in $\nabla_r$ in order—the same order as they appeared in the stream, so random—until $\vec{t}_r$ is eliminated, or is compared against them all and found to be maximal.*

Best *compares $\vec{t}_r$ against maximals (and only maximals, since $\vec{t}_r$ is the last tuple) until $\vec{t}_r$ is eliminated, or $\vec{t}_r$ is the selected maximal in some pass. The order of comparisons is the order in which* Best *finds the maximals, one per pass. This order is not random, but correlated by volume descending, by Lem. 13.*

*By Lem. 14,* Best *then makes fewer comparisons for $\vec{t}_r$, on average, than does* BNL-*open. By Thm. 12,* BNL-*open is $\mathcal{O}(kn)$. Thus so is* Best*.* □

Given that BNL is well behaved if its window is unbounded and if its window is bounded to just one, does BNL remain well behaved for fixed window sizes in between? We do not know, although we conjecture it would still be $\mathcal{O}(kn)$ in average case for any window size.

**Conjecture 16** *Under CI,* BNL *with any fixed window size ($w \geq 1$) and using the append window policy has an average-case runtime of $\mathcal{O}(kn)$.*

The difficulty behind proving this is that it is now quite hard to characterize what is in the window at any given stage. Once the window becomes full, any stream tuple that is incomparable with all the window tuples is written to an overflow file, since there is no space to append it to the window. A stream tuple that *dominates* some some window tuples, however, will be added to the window, since its eliminations create room. Furthermore, this means the window may no longer be full (as the stream tuple may have eliminated several window tuples), so the next *incomparable* stream tuples *can be* added.

In fact, BNL is observed to be badly behaved with respect to window size. One would expect that the more resources that the algorithm is allocated—in this case, more main memory—the better it would perform. However, the opposite is true. It has been shown experimentally in [17] that Best makes far fewer comparisons than BNL (with an unbounded window). During our development of SFS [15],[16] we found that BNL's performance worsened significantly as we increased its window allocation. Of course, we know now that BNL with an unbounded window (under the append policy) is $\mathcal{O}(kn)$, by Thm. 12. So while BNL with an unbounded window performs worse than Best, it can only be worse by some constant factor. It would be interesting to observe experimentally whether BNL under certain fixed window sizes are seen to perform significantly worse than BNL with an unbounded window.

While BNL does not need the input set to be sorted, its good performance relies on the input set being randomly ordered. Unfortunately, data is often ordered. For instance, data in a database system is often ordered as it is usually indexed in various ways. It was shown in [15] that, if the data set is already ordered in some way, but not for the benefit of finding the maximals as in SFS, BNL can perform very badly. In fact, it is easy to prove that under certain stream orders and for a fixed window size, BNL would have $\mathcal{O}(kn \lg n)$ average-case performance, or potentially worse.

Let us next consider SFS. The SFS algorithm consists of two phases: in phase one, the input set is sorted; and, in phase two, the sorted stream is filtered for the maximals. Of course, SFS is immune to any original ordering on the input since it sorts the data itself. Let us sort the data as a nested sort on $d_{k-1}, \ldots, d_0$ descending. This is equivalent to what the SQL statement order by $d_{k-1}$ desc, …, $d_0$ desc would provide. Of course, since we are assuming sparseness—and so there are no repeated values on $d_{k-1}$—this is equivalent to just order by $d_{k-1}$ desc. We assume that

[16]The authors of this paper and of the SFS work [15]—Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang—overlap.

the window management strategy compares against window tuples in a random order (*random walk*), and we assume for now an unbounded window.

The bounds on **mttf** introduced for BNL are the same here. When considering $\vec{t_i}$, we need to consider the ratio of the number of maximals in the dominating space above $\vec{t_i}$ over the number of maximals in the window collected so far, $\nabla_i$. However, we note that all the tuples that precede $\vec{t_i}$ have a higher $d_{k-1}$ value. This reduces the degrees of freedom by one in the estimation formula for how many comparisons for $\vec{t_i}$ are expected:

$$\int_{x_{k-2}=0}^{1} \ldots \int_{x_0=0}^{1} \frac{\mathrm{lh}^{\,k-1} i}{\mathrm{lh}^{\,k-1} x_0 \ldots x_{k-2} i} dx_0 \ldots dx_{k-2}$$

So the overall number of comparisons per vector SFS is expected to make is bounded by

$$\int_{z=0}^{1} \int_{x_{k-2}=0}^{1} \ldots \int_{x_0=0}^{1} \frac{\mathrm{lh}^{\,k-1} zn}{\mathrm{lh}^{\,k-1} x_0 \ldots x_{k-2} zn} dx_0 \ldots dx_{k-2} dz$$

We could not safely choose *append* as a window policy for this version of SFS as we did for BNL. It would *not* perform well. The list of maximals (in the order they are found) is not independent with respect to the sort order of the stream. As the value on $d_{k-1}$ decreases, the average of $d_0, \ldots, d_{k-2}$ *of the maximals* increases. The dimensions with respect to the maximal set are anti-correlated. This maximal list would be correlated with that from least dominating to most. (So a *prepend* policy in this case would, in fact, work well.) Generally, an analysis of average-case performance is made significantly more complex when we must consider an "intelligent" window policy. Thus, consider a *random-walk* policy: a candidate is compared against window tuples in a random order. Then it does not matter whether the tuples in the window have an inherent order or not. In the case of BNL, the append policy we used with it is effectively the same as using a random-walk policy with it. In either case, BNL would compare the candidate against window tuples in a random order.

While SFS with the stream sorted by one of the dimensions reduces the degrees of freedom by one, the estimated number of comparisons per vector is still based on a dimensionality of $k$. An improvement on the SFS algorithm in this case is to change the window strategy to *eliminate* tuples. Since we sorted on $d_{k-1}$, we only need to maintain the maximals with respect to $d_0, \ldots, d_{k-2}$ of the tuples added to the window. Thus, if a new maximal $\vec{t_i}$ is added to the window, any maximals in the window that it dominates *with respect to* $d_0, \ldots, d_{k-2}$—so not considering $d_{k-1}$—can be removed. The expected number of comparisons made against stream tuple $\vec{t_i}$ and the window is now bounded by $\mathrm{lh}^{\,k-2} zn/\mathrm{lh}^{\,k-2} x_0 \ldots x_{k-2} zn$ instead of $\mathrm{lh}^{\,k-1} zn/\mathrm{lh}^{\,k-1} x_0 \ldots x_{k-2} zn$. So the expected number of comparisons per vector is bounded by

$$\int_{z=0}^{1} \int_{x_{k-1}=0}^{1} \ldots \int_{x_0=0}^{1} \frac{\mathrm{lh}^{\,k-2} zn}{\mathrm{lh}^{\,k-2} x_0 \ldots x_{k-2} zn} dx_0 \ldots dx_{k-2} dz$$

This is precisely one dimension better than BNL. This version of SFS can be considered as a variation of BNL which sorts the data first along one dimension. The advantages are a reduction of one in degrees of freedom ($k$) *and* that whenever a tuple is added to the window, it is already known to be maximal and can be reported.

We can safely choose *append* as the window policy this time for SFS with an elimination policy, as we did for BNL. Under the assumptions of CI, the local maximals with respect to $d_0, \ldots, d_{k-2}$, up to any given stage, are distributed randomly and uniformly over the stream sorted on $d_{k-1}$. Therefore, the order they collect in the window is random.

**Theorem 17** *Under CI,* SFS *with an unbounded window size, sorting by one dimension descending, and with append* with *elimination as the window policy, has an average-case runtime of* $\mathcal{O}(kn)$.

**Proof 17** *This follows from the proof for Thm. 12.*
□

If we change the window policy for either BNL or SFS, or we change the sort order for SFS, then a new analysis may be needed. It is not guaranteed that any variant of BNL or SFS will have $\mathcal{O}(kn)$ average-case runtime performance. We must be careful in our design choices.[17]

For SFS, consider ordering on a *volume* (or *entropy*) measure as in [15, 16]. For the window policy, consider the append policy, so the window is ordered by volume also. The proof of Thm. 17 does not apply to this version of SFS. For average-case analysis of SFS under these assumptions, we need to know how many of the maximal points dominate any given non-maximal. For any maximal point, it is compared against every maximal point before it in stream. There are $m(m-1)/2$ 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 18** *Under UI (Def. 2), in the limit of n, the probability a non-maximal point is dominated by the maximal point with the highest volume converges on one.* [20]

**Proof 18** *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 [5]. Consider the (virtual) point* $\vec{v}$ *with coordinates* $\vec{v}\,[i] = 1 - ((\ln n)/n)^{1/k}$*, for each* $i \in k$*. The probability that no point from the data set dominates* $\vec{v}$ *then is* $(1 - (\ln n)/n)^n$*, which is at most* $e^{-\ln n} = 1/n$*.*

[17]Note that best-case and worst-case for SFS as discussed in §3.1 are unaffected by the sort order (as long as the sort is a valid topological sort).

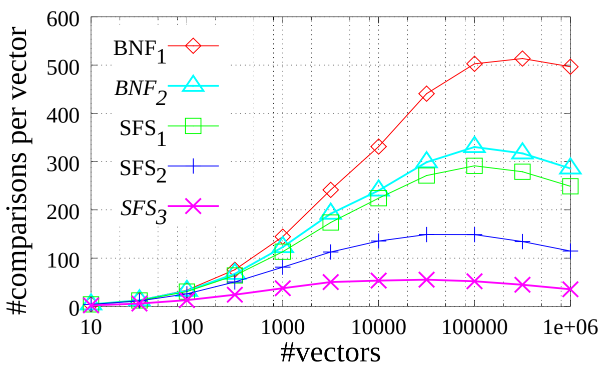


Figure 8: Number of comparisons per vector ($k = 7$).

*The expected number of points dominated by $\vec{v}$ (and hence, dominated by any point that dominates $\vec{v}$) 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 $\vec{v}$'s (which would include any points that dominate $\vec{v}$) 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 19** *Under UI (Def. 2),* SFS *sorting the stream by volume descending and using the append window policy, for any fixed window size, has an average-case runtime of $\mathcal{O}(kn + n\lg n)$.* [20]

**Proof 19** *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, $m(m-1)/2$ maximal-to-maximal comparisons are 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 $\mathcal{O}(kn)$.*

*We use the append window policy. This means that the window contains maximals in volume descending order, the same order from the stream. No tuples are eliminated from the window, except when the window is cleared between passes. Any stream tuple written to overflow will be compared against the next batch of maximals in the next pass. Thus, the window size cannot affect the number of comparisons.* □

By simulation, performances for two variants of BNL and three variants of SFS are shown in Figure 8. Data sets of ten vectors to one million vectors (by powers of ten) were generated, and the variants were simulated to count the number of comparisons per vector made. At least thirty trials were run in each bracket. The data was normalized and the order random.

- $\mathsf{BNL}_1$: BNL with the append window policy as for Thm. 12.
- $\mathsf{BNL}_2$: BNL with a window policy that keeps the window tuples sorted by volume descending, with respect to $d_0, \ldots, d_{k-2}$.
- $\mathsf{SFS}_1$: SFS sorting the input on one dimension descending and with the random-walk window policy without elimination.
- $\mathsf{SFS}_2$: SFS sorting the input on one dimension descending and with the append window policy with elimination as for Thm. 17.
- $\mathsf{SFS}_3$: SFS sorting the input on one dimension descending and with a window policy that keeps the window sorted by volume descending. (No elimination is done in the window, but the comparison against the window can terminate early.)

The variants $\mathsf{BNL}_1$, $\mathsf{SFS}_1$, and $\mathsf{SFS}_3$ are the idealized versions we used for analyses. The variants $\mathsf{BNL}_2$ and $\mathsf{SFS}_2$ are realistic versions of the algorithms with seemingly good choices made for sort order and window policy to improve performance. For $\mathsf{BNL}_2$, we keep the window sorted by the local maximals's volume measure. A candidate is then compared against window tuples in order. Tuples near the beginning of window are more likely to dominate. In [8], they keep a counter with each window tuple. Any time a candidate is dominated by a given window tuple, the corresponding counter is incremented. The window is then kept sorted by the counters's values. Thus, they attempt to measure dynamically a tuple's dominance capacity.

$\mathsf{SFS}_2$'s performance is precisely what we would see for BNL for data sets of dimensionality six ($k = 6$). $\mathsf{SFS}_3$ represents one of the more effective variants of SFS we have found: the data is sorted on one dimension (as by order by); the window is kept sorted by tuples's volume measures; and a candidate is compared against the window tuples in order. As an extra optimization, once the candidate's volume measure exceeds the next window tuple's, we stop the comparisons. The candidate is clearly a maximal. None of the rest of the window tuples could dominate it, since each has a lower volume measure. This effectively curtails many maximal-to-maximal comparisons to well below $m(m-1)/2$. Thus $\mathsf{SFS}_3$ performs remarkably well in comparison with the others.

The graph in Figure 8 does not reflect SFS's sort phase. Even so, note that even adding the sorting expense for SFS, it outperforms BNL in number of comparisons. For a million records, SFS would need to do roughly 20 comparisons extra per vector in the sort.[18] Adding this, SFS still does far less comparison work than BNL. Of course, in the extreme limit, SFS must do more comparison than BNL, on average. Furthermore, SFS could lose to BNL in implementation; there are other expenses besides CPU-load to consider, such as I/O-load. (In fact, for the experiments in this paper in §3.4.2, this is the case.)

One must note that all of these are still quite expensive algorithms, in practice. It is surprising that each would only make *one* comparison per vector, on

[18]Note that $\lg_2 10^6 \approx 20$ and that merge-sort is close to ideal on the concrete number of comparisons made.

average, in the extreme limit of $n$.[19] However, for realistic input sizes ($n$), the number of comparisons made per vector is appreciably high. (Note that the graph in Figure 8 is in log-scale along the x-axis.)

Runtime for these algorithms is quite affected by the dimensionality. Each extra dimension added is, after all, an additional degree of freedom, thus the problem becomes correspondingly more complex and more expensive to solve. This should only be expected.

BNL is a remarkable algorithm. Still, we have seen that there are issues with its performance, and improvements can be made. By sorting, SFS effectively reduces the dimensionality of the problem by one (which can be a substantial improvement on its own), and addresses many of the weaknesses of BNL. However, the sort step gives SFS too high an average-case runtime. We would like to have the best of both: SFS's improvements, but still $\mathcal{O}(kn)$ overall average-case runtime performance.

## 3.4 The LESS Algorithm

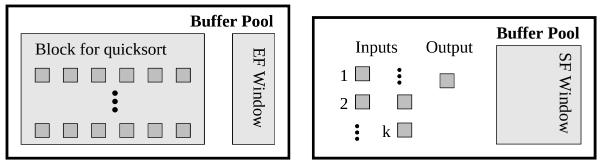
### 3.4.1 Description

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 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 (volume) scores, as discussed in §3.1 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 (Figure 9(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.[20]



(a) in pass zero (b) in pass $f$

Figure 9: Buffer pool for LESS.

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 (Figure 9(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.

[19]Note that we do not actually know this for the realistic variations $\mathsf{BNL}_2$ and $\mathsf{SFS}_3$ here, but one would assume so for them also.

[20]We use a practical variation on this EF policy in the version ran in the experimental evaluation in the next section. There, the EF window keeps the maximals of what has been seen so far, just as BNL does in its filtering for the skyline.

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 §3.4.3, 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 $\mathcal{O}(n)$.

### 3.4.2 Experimental Evaluation

To implement LESS and other skyline algorithms for benchmarking, we implemented a code base that we nicknamed Shiprec.[21] An implementation of LESS requires integration with an external sort routine which, in turn, requires at least simplified buffer pool and diskspace managers. Shiprec is implemented to use page-based, nonblocking reads and writes, and double-buffering. It is implemented in C for the *gcc* compiler. It uses the *pthreads* library. A thread "watchdog" is attached to each active buffer pair to implement non-blocking I/O. A large file is allocated via the operation system (Linux) via *dd* in advance to serve as the program's diskspace.

The experiments were run on an Intel-based computer running Linux with a 2.4.32 kernel. The machine has a single CPU, an Intel Pentium 4 (with 8k L1 and 512k L2 cache) clocked at 3.20GHz. It has one GB main memory. The disk controller and bus are IDE-SCSI, and the machine has two standard ATA disks.

[21]The initial Shiprec package was part of the work that Ryan Shipley (one of the authors) did for his undergraduate honors thesis at the College of William and Mary in 2003 under the supervision of Parke Godfrey (another of the authors).

The experiments are run on 5,000,000 record sets with respect to 5 through 9 dimensions.[22] Each record is 100 bytes, so the total size of a record set is 500 million bytes. This size helps defeat page caching by the operating system—as the machine has one giga-byte of main memory—to ensure the I/O cost is accurately reflected.

The record sets are generated by Shiprec. Column values are chosen randomly, and the record sets obey the UI criteria from Def. 2. Each column used as a skyline criterion has an integer value on $1 \ldots 2^{15}$. The disk-page size in Shiprec was set to 8,192 bytes.

There are many parameters that can be adjusted for these algorithms within Shiprec, each of which can affect performance. A key parameter, of course, is the buffer pool allocation made to the algorithm. LESS also uses an EF window during its quicksort pass. Its size needs to be set.

If the EF window is too large, it will take more time simply as management of the EF window starts to have an impact. On the other hand, if the EF window is too small, the algorithm might 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 made little difference to LESS's overall performance, with some small trade-offs in comparisons and I/O usage. (We make clear why this should be the case in §3.4.3.) We set the EF window at a single page (80 records) for what we report here.

We experimented with various buffer pool allocations, from 10 to 500 pages. The size affects primarily the efficiency of the sorting phase, as expected. BNL is also quite sensitive, and ill behaved, with respect to buffer pool allocation. In many cases, it is *slowed down* with a larger allocation, due to a large increase in the number of comparisons the algorithm performs. We set the allocation at 100 pages as a reasonable choice for what we report here. (As double buffering is used, this accommodates up to a 50-way merge or a 50-page block operation, such as quicksort. With a 8,192 byte page size, this is 819,200 bytes of main memory available to the program for data operations.

Figures 10 and 11 report timing and I/O results, respectively. All experiments were run using our Shiprec code as described above. ExtSort represents just running the external sort procedure on the record set; the maximals are not computed. The records are sorted descending on an *entropy* field that is generated when the records are initially read. SFS is an implemen-

[22]These are not the same experiments that are reported in [20]. Each input set there had 500,000 records for a size of 50 million bytes. We were asked to run these over larger record sets. The machine used for these experiments is also different than before. We also made minor improvements to our experimental platform, Shiprec, including implementing BNL within it for comparison.

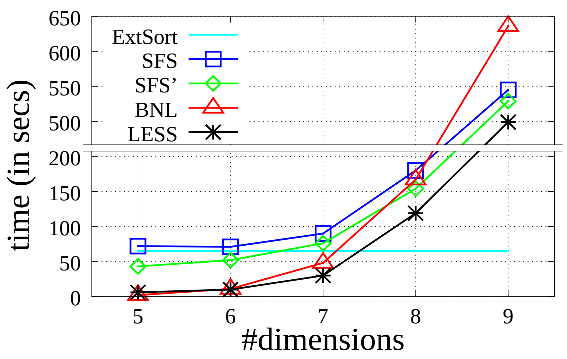


Figure 10: Time.

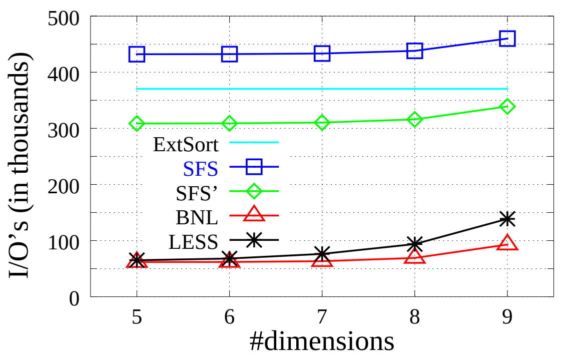


Figure 11: I/O's.

tation of the SFS algorithm as described in §3.1. For SFS, the records are initially sorted descending by their entropy values (as by ExtSort). The skyline window policy used is *append*. SFS′ makes one of the two major changes from SFS to LESS: the last merge pass of the sort is combined with the first skyline pass. This saves I/O's but does not reduce any of the comparison load. LESS is an implementation of the LESS algorithm within Shiprec as described in the previous sub-section. Again, the records are sorted by entropy descending. The skyline window policy used is *append*.

BNL is an implementation of the BNL algorithm within Shiprec as described in §3.1. The window policy used is a variation on *append*. The window list is walked in order for each stream record. If the stream record is found to dominate a window record, it replaces that record in that list location within the window; otherwise, it is appended to the window list (if there is room). This has a better profile for BNL than simple append, similar to that of $\mathsf{BNL}_2$ discussed in §3.3. No sorting is done for BNL, of course; the input stream is randomly ordered.

In each case, the results are written back to disk and this cost is reflected in the measures. The record set is already generated and on disk before the clock is started, so this cost is not reflected (nor should it be).

For these settings and the Shiprec implementation, the external sort takes 67 seconds, on average. This is regardless of the number of dimensions considered as the sort is with respect to a single field (entropy). This is a lower bound then on SFS's performance. For 5 and 6 dimensions, one can see the improvement of SFS′ as it does better than the external sort. This is because the skyline filtering has started during the last merge pass and many records are eliminated. This results in I/O savings. In all the experiments, the I/O cost for SFS′ is better than for ExtSort, which, in turn, is necessarily better than for SFS. The number of comparisons for SFS and SFS′ are the same, however.

LESS shows a large performance improvement over SFS and SFS′. The filtering by the EF window during the quicksort pass of the external sort is not only theoretically beneficial—this renders LESS to be $\mathcal{O}(kn)$ as proved in the next sub-section—it is also practically beneficial. This reduces LESS's I/O load significantly compared with SFS. Shiprec's LESS computes the skyline for the 5,000,000 records on 7 dimensions in 30 seconds.

Shiprec's BNL performs quite well under these parameters, particularly for lower dimensionality. On the data set on 5 dimensions, it computes the skyline in 2 seconds while LESS takes 6 seconds. The two tie at around 10 seconds for the skyline on 6 dimensions. On 7 dimensions, BNL has fallen behind to 48 seconds with LESS taking 30 seconds. On 8 dimensions, it performs no better than SFS′. On 9 dimensions, it performs significantly worse than the rest. It has the best I/O cost of all the algorithms. Its computational load (how many comparisons are performed), however, is worse than the rest. This profile fits well with our analytic understanding of the algorithms.

BNL is also ill-behaved. If the buffer pool allocation is increased here, BNL's performance worsens as it comparison load goes up. For the other algorithms, their performance improves step-wise, as expected, as the buffer pool allocation is increased. Mainly, at some point, the external sorting can be done in fewer passes, reducing the I/O load. SFS and LESS do not experience an increase in comparison load as the allocation is increased. BNL is also sensitive to the order of the input stream. If the input stream is sorted (or correlated with) a topological order that is opposite the skyline order, BNL will thrash. (This was demonstrated in [15].) For instance, Shiprec's BNL run for 5 dimensions from Figure 10 for which the data is randomly ordered took 2 seconds. When run on a 5 million record set that has been ordered by one of the dimension fields from lowest to highest—the skyline criteria here are for highest—BNL takes 418 seconds. Of course, SFS and LESS are immune to input order.

All the processes show signs of being CPU-bound even for computing the 7 dimensional skyline. Although SFS′, BNL, and LESS have I/O costs below ExtSort, by 8 dimensions, all have much worse times than ExtSort, meaning that the computational load is out-stripping the I/O load. BNL and LESS have proportionally much less I/O cost than ExtSort, but their times approach ExtSort for 7 dimensions, demonstrating CPU-boundedness.

The results also demonstrate just how expensive each additional dimension to the skyline problem truly is in practice. This primarily arises due to the fact that the final skyline set is larger for higher dimensionality. For instance, for the data sets used here, the final skyline sets were 2,840, 9,126, 25,715, 59,171, and 128,058 records for 5 through 9 dimensions, respectively.[23] All the algorithms effectively must compare each skyline record against every other one to verify it.

There are many improvements that could be made to our Shiprec code base that would improve performance of the algorithms. Shiprec does not exploit sequential reads and writes. Doing so could improve on the cost of the external sort operations significantly. We have seen that Shiprec pays a fair overhead in thread management. In part, this is due to inefficiencies in thread management by the Linux 2.4 kernel. (The 2.6 kernel reportedly improves greatly on this.) It is also due to our design: Shiprec maintains a thread for *each* active buffer pair. If a 200-way merge is being performed, that means 200 watchdog threads. In retrospect, this is wasteful and not needed. One could use a small pool of threads and semaphore for this purpose. This design also limits Shiprec's buffer pool management. Linux 2.4 has a limit of around 250 threads allowed per process. This effectively meant we could not test with over 500 buffer frames. A number of processing steps in Shiprec are naïve and could be improved. (For example, when performing a multi-merge, it polls for the next record rather than using a priority queue.) Lastly, Shiprec can only address up to two gigabytes in diskspace as limited by a 32-bit integer. This limited us to testing on up to half a gigabyte data sets, as in these experiments. (For some of the algorithms, it can handle up to a gigabyte data set.) We would like to test on much larger record sets.

A commercial caliber implementation of these algorithms, and the next generation of our Shiprec platform, could improve performance of LESS and SFS further significantly. Some improvements could no doubt benefit BNL also, but likely to a lesser degree. For instance, Shiprec's thread overhead did not affect BNL much; it does not use multi-way merges.

### 3.4.3 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.

[23]These sizes match well with the analytically predicted values for a five million record set under CI.

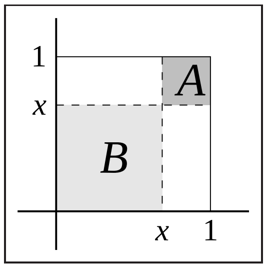


Figure 12: Choosing point $v$.

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 [5] to prove FLET's $\mathcal{O}(n)$ average-case runtime performance and in Lemma 18.

**Theorem 20** *Under UI (Def. 2),* LESS *has an average-case runtime of* $\mathcal{O}(kn)$. [20]

**Proof 20** *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 Figure 12 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* $\mathcal{A}$*?*[24] *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*

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

*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. 17). 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 21** *Under CI (Def. 2) and the model in §2.4,* LESS *has a best-case runtime of $\mathcal{O}(kn)$.* [20]

**Proof 21** *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 22** *Under CI (Def. 2),* LESS *has a worst-case runtime of $\mathcal{O}(kn^2)$.* [20]

**Proof 22** *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. 8).* □

## 3.5 Issues and Improvements

Since our experiments in §3.4.2, we have been focusing on how to decrease the CPU load of LESS, and of maximal-vector algorithms generally. LESS and SFS must make $m(m-1)/2$ comparisons just to verify that the maximals are, indeed, maximals. 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, §3.2 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. SFS$_3$ in Figure 8 and discussed in §3.3 does this.) 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 by roughly a quarter.

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 represents the worst-case scenario for maximal-vector algorithms (§2.4). We are able to handle reasonably well some cases of anti-correlation, for instance, when one dimension is highly anti-correlated with another one. We may 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.

## 3.6 Lifting the Assumptions

The example in §2.1 violates the assumptions made in §2.4 that we used for average-case analyses. The prices of the hotels do not seem *uniformly* distributed, and likely they are not. The number of stars (quality) of hotels is not *sparse*. There are only several possible values. It is likely that number of stars and price are correlated, and price and distance to the beach are quite possibly anti-correlated, thus the dimensions are not *independent*. Most real-world data—and skyline queries over that data—will violate those assumptions. How important in truth are they?

In many cases, we did not need to assume uniformity. The assumption seemed necessary in other cases. The algorithms FLET and LESS relied on it, and SFS benefits from it. We can actually remove the assumption for these.

FLET needs to choose a virtual point that it will use to eliminate points on the initial pass. The coordinates of this virtual point can be trivially determined if the data set is normal. Note that we simply need to know a rank-value along each dimension: that so many vectors rank above that given value. This *selection* problem is known to be linear, however. One can find the element of a given rank from an unordered list in $\mathcal{O}(n)$ steps [7].

We could determine therefore the value corresponding to the target rank along each dimension in $\mathcal{O}(kn)$ work. (This could be accomplished in fewer than six passes over the data.) Thus, with $\mathcal{O}(kn)$ preprocessing, we remove the need for uniformity for FLET and LESS.[25]

In a database system, one often has statistics about the data available. In many cases, these can be used to approximate the normalization mapping for the data.

When the sparseness (distinct-value) assumption is violated—so there are 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, points that are equivalent on all the dimensions) [19]. We can prove that if there are few duplicates over the vectors, but repeated values along the dimensions, the expected number of maximals *decreases*. Since our average-case for maximal-vector algorithms is predicated on $\widehat{m}$, all will perform at least as well when the data is dense.

The case when there are many duplicates in the input is different. Then the number of maximals may be much greater than $\widehat{m}$ under CI: the same maximal is repeated many times. For an industrial-caliber skyline algorithm, it would be possible to address this case. LESS, for instance, does not need to keep the tuples in the window. It only needs to keep a projection of the tuple on the skyline columns. Furthermore, there is no need to maintain duplicates in the window. Thus, if an optimizer predicts a duplicate flood, LESS could first find all the distinct maximal projections, hash these in main memory, and on a subsequent pass of the data, select the maximal tuples.

The only intrinsically difficult assumption is independence. Maximal-vector algorithms perform poorly when there are many maximals, and this occurs when the dimensions have anti-correlations among them. Whether better algorithmic approaches exist to handle maximal-vector computation for highly anti-correlated cases remains open.

## 4 Related Work

In recent years, there has been much interest in maximal vector computation in the database community. In this paper, we have focused on generic algorithms to find the maximal vectors. However, research in this area also covers index-based algorithms, and extensions to the standard maximal vector problem.

The goals of index-based algorithms 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 [24] exploits R-trees and modifies nearest-neighbors approaches for finding skyline points progressively. This work is extended upon in [27, 28] in which they apply branch-and-bound techniques to reduce significantly the I/O overhead. In fact, the algorithm in [27] is shown to deliver better performance than any previous (prior to 2003) index-based algorithms. In [18, 30], 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. No performance comparison with other algorithms is provided, however.

We note that the generic SFS algorithm can be modified to become an index-based algorithm. The result of the first phase of the algorithm, the nest-sorted list of tuples, can be maintained using a B+ tree index. The second phase of the algorithm, the actual Skyline computation, can then be performed against that index.

Index-based algorithms for computing the skyline (the maximal vectors) have serious limitations. The performance of indexes—such as R-trees as used in [24, 27]—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.

We also note 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

$$\begin{aligned}&\mathsf{maxes}_{\{\mathsf{d}_0,\ldots,\mathsf{d}_{i-1}\}}(\mathbf{T}) \cup \mathsf{maxes}_{\{\mathsf{d}_i,\ldots,\mathsf{d}_{k-1}\}}(\mathbf{T})\\ &\quad\subsetneq \mathsf{maxes}_{\{\mathsf{d}_0,\ldots,\mathsf{d}_{k-1}\}}(\mathbf{T})\end{aligned}$$

Furthermore, if the sparseness (distinct-values) assumption from §2.4 is lifted, the union is no longer even guaranteed to be a subset. (This is due to the possibility of ties over, say, $\mathsf{d}_0, \ldots, \mathsf{d}_{i-1}$.)

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. (In [13], cases of commutativity of skyline with other relational operators are shown.) For any skyline query that involves a select condition on an attribute which is not a skyline attribute, an index that would have applied to the query without the select will not be applicable.

Recall the criteria for good skyline algorithms [28] discussed in §2.3. To satisfy criterion one, *progressiveness*, some preprocessing is required: index creation for index-based algorithms, or sorting for SFS or LESS.

[25] We would not want to do this in practice. This does show, however, that FLET and LESS theoretically have $\mathcal{O}(kn)$ average-case runtime under CI.

This preprocessing is a one-time effort and can be used in subsequent queries, provided that the maintained structure is easily updateable. All of the discussed algorithms satisfy the second criterion, *absence of false hits*, as they compute the exact, not an approximate, skyline. Also, all algorithms, except BNL, satisfy criterion three, *absence of temporary false hits*. The nearest-neighbor algorithm of [24] violates the fourth criterion, *fairness*, as it returns skyline points according to their minimum coordinates in some dimension. The algorithms in [1, 24, 28] and the LESS algorithm described in this paper are the only algorithms that can incorporate user preferences (criterion five) to order skyline points. As discussed above, none of the index-based algorithms scales well with respect to the number of dimensions, and thus violate the sixth criterion, *universality*.

There has been much work recently devoted to extending research in skyline computation to new problems and domains. Some of this work includes:

- semantics and computation of skyline in subspaces [29, 31];
- computation of skyline cubes [34];
- skyline computation over partially ordered domains [3, 10, 11];
- skyline computation over sliding windows [26];
- skyline computation in mobile environment [22];
- skyline of categorical data [2];
- approximate skyline [23]; and
- estimation of skyline cardinality [12, 19].

## 5 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. We proved that the scan-based skyline algorithms, while seemingly more naïve, are much better behaved. We introduced a new algorithm, LESS, which improves over the existing skyline algorithms, and we prove that its average-case performance is $\mathcal{O}(kn)$.

There remains room for improvement, and there are clear directions for future work. While we can construct algorithms that are asymptotically good without the uniformity assumption, with it we can improve performance. We want to understand how to improve performance in similar ways without needing to assume uniformity. 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 §3.5.

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.

## References

[1] W.-T. Balke and U. Güntzer. Multi-objective query processing for database systems. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, pages 936–947, Toronto, Canada, Aug. 2004. Morgan Kaufmann.

[2] W.-T. Balke and U. Güntzer. Supporting skyline queries on categorical data in web information systems. In *IASTED International Conference on Internet and Multimedia Systems and Applications (IMSA 2004)*, pages 1–6, 2004.

[3] W.-T. Balke and U. Güntzer. Efficient skyline queries under weak pareto dominance. In *IJCAI-05 Multidisciplinary Workshop on Advances in Preference Handling (Preference 2005)*, pages 1–7, 2005.

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

[5] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 179–187. ACM/SIAM, Jan. 1990.

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

[7] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Science*, 7(4):448–461, 1973.

[8] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th ICDE*, pages 421–430, 2001.

[9] C. Buchta. On the average number of maxima in a set of vectors. *Information Processing Letters*,

33:63–65, 1989.

[10] C. Y. Chan, P.-K. Eng, and K.-L. Tan. Efficient processing of skyline queries with partially-ordered domains. In *ICDE*, pages 190–191, 2005.

[11] C. Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *SIGMOD Conference*, pages 203–214, 2005.

[12] S. Chaudhuri, N. Dalvi, and K. Raghav. Robust cardinality and cost estimation for skyline operator. In *ICDE*, 2006. To appear.

[13] J. Chomicki. Querying with intrinsic preferences. In C. S. Jensen, K. G. Jeffery, J. Pokorný, S. Saltenis, E. Bertino, K. Böhm, and M. Jarke, editors, *Proceedings of the 8th International Conference on Extending Database Technology (EDBT)*, LNCS 2287, pages 34–51, Prague, Czech Republic, 2002. Springer.

[14] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. Technical Report 04, Computer Science, York University, Toronto, Ontario, Canada, Oct. 2002.

[15] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 717–719, 2003. See [14] for a longer version.

[16] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting: Theory and optimization. In M. A. Klopotek, S. T. Wierzchon, and K. Trojanowski, editors, *Proceedings of the Intelligent Information Systems Conference (IIS): New Trends in Intelligent Information Processing and Web Mining*, Advances in Soft Computing, pages 593–602, Gdansk, Poland, June 2005. Springer.

[17] P. Ciaccia. Evaluating preferences with non-transitive preferences. Presentation at the Dagstuhl Seminar 04271 (Preferences: Specification, Inference, Applications), June 2004.

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

[19] P. Godfrey. Skyline cardinality for relational processing. In D. Seipel and J. M. T. Torres, editors, *Proceedings of the Third International Symposium on Foundations of Information and Knowledge Systems (FoIKS)*, pages 78–97, Wilhelminenberg Castle, Austria, Feb. 2004. Springer.

[20] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-Å. Larson, and B. C. Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005)*, pages 229–240, Trondheim, Norway, Aug. 2005. ACM.

[21] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis: The control project. *IEEE Computer*, 32(8):51–59, 1999.

[22] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in manets. In *ICDE*, 2006. To appear.

[23] W. Jin, J. Han, and M. Ester. Mining thick skylines over large databases. In *PKDD*, pages 255–266, 2004.

[24] D. Kossmann, F. Ramask, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB-2002)*, pages 275–286, Aug. 2002.

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

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

[27] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 467–478. ACM Press, 2003.

[28] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[29] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *VLDB*, pages 253–264, 2005.

[30] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pages 301–310, Rome, Italy, 2001. Morgan Kaufmann.

[31] Y. Tao, X. Xiao, and J. Pei. SUBSKY: Efficient computation of skylines in subspaces. In *ICDE*, 2006. To appear.

[32] R. Torlone and P. Ciaccia. Finding the best when it's a matter of preference. In P. Ciaccia, F. Rabitti, and G. Soda, editors, *10th Italian National Conference on Advanced Data Base Systems (SEBD 2002)*, pages 347–360, June 2002.

[33] R. Torlone and P. Ciaccia. Which are my preferred items? In *Workshop on Recommendation and Personalization in eCommerce (RPEC)*, pages 1–9, Malaga, Spain, 2002.

[34] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.