Scalable Parallelization of Skyline Computation for Multi-core Processors

Sean Chester*, Darius Šidlauskas†, Ira Assent*, and Kenneth S. Bøgh*

*Data-Intensive Systems Group, Aarhus University, Denmark

† Data-Intensive Applications and Systems Laboratory, EPFL, Switzerland

schester@cs.au.dk darius.sidlauskas@epfl.ch ira@cs.au.dk ksb@cs.au.dk

***Abstract*—The skyline is an important query operator for multi-criteria decision making. It reduces a dataset to only those points that offer optimal trade-offs of dimensions. In general, it is very expensive to compute. Recently, multicore CPU algorithms have been proposed to accelerate the computation of the skyline. However, they do not sufficiently minimize dominance tests and so are not competitive with state-of-the-art sequential algorithms.**

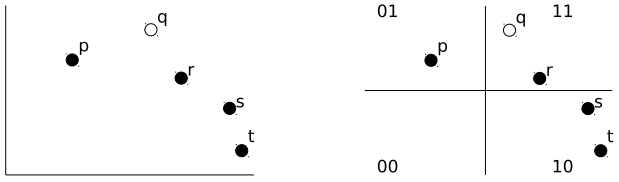
In this paper, we introduce a novel multicore skyline algorithm, Hybrid**, which processes points in blocks. It maintains a shared, global skyline among all threads, which is used to minimize dominance tests while maintaining high throughput. The algorithm uses an efficiently-updatable data structure over the shared, global skyline, based on point-based partitioning. Also, we release a large benchmark of optimized skyline algorithms, with which we demonstrate on challenging workloads a** 100**-fold speedup over state-of-the-art multicore algorithms and a** 10**-fold speedup with** 16 **cores over state-of-the-art sequential algorithms.**

I. Introduction

Skyline computation, introduced in 2001 [4], is still an active research area with applications in route planning for road networks [14], [21], data exploration [5], web service composition [1], and many other multi-criteria decision-making domains wherein (possibly conflicting) preferences need to be balanced. Figure 1a illustrates the skyline over an example dataset. If small values are preferred (e.g., the points represent fuel consumption and expected travel time), then q is clearly a worse option than (i.e., is *dominated by*) p, since it has larger values for both coordinates. The skyline consists of all non-dominated points (in this case, p, r, s, t).

However, the skyline is expensive to compute, especially when it is large relative to the input, because each skyline point (at least implicitly) needs to be compared to every other skyline point. This computational challenge has prompted the use of modern computing platforms, such as GPUs [3], [8] and multicore CPUs [13], [16], as well as distributed environments [12], including MapReduce [17], [19], [22], to accelerate the computation. Of these, multicore CPUs are a particularly attractive option, because the cost of shared data structures is much lower and parallel work need not be isolated. Still, we demonstrate the surprising conclusion that current multicore skyline algorithms can be outperformed *by at least an order of magnitude* by sequential algorithms on modest workloads.

Current multicore algorithms adopt the same paradigm as distributed algorithms, a divide-and-conquer approach wherein the dataset is cut, local skylines are computed in isolation by each thread, and then local results are merged to produce a global result. This paradigm suffers two principal drawbacks. First, if the local results are large, then the merging step becomes prohibitively expensive. Moreover, this partitioning *hinders* pruning capacity. If p and q in Figure 1a are allocated to separate threads, then the dominance cannot be detected until the more expensive merge phase, once all threads complete.



(a) Skyline example (b) Partitioning

Fig. 1: (a): p, r, s, t are in the skyline, but not q: it has higher x- and y-values than p. (b): partitioning reveals incomparability and refines the probability of a point dominating another.

By contrast, we adopt a different paradigm, where all known skyline points are maintained in a global data structure. The skyline is updated at regular synchronization points and read by all threads. We order the skyline points in the data structure to maximize the probability of detecting new dominance relationships quickly so that subsequent dominance tests can be averted. The processing of points is done in ordered blocks that guarantee each point is compared to at most α more points than in a sequential algorithm, a guarantee that a divide-and-conquer approach cannot offer.

As a consequence, we have no expensive merge phase, dominance relationships are determined early rather than being severed by file cutting, and, like state-of-the-art sequential algorithms, we can report results progressively. In all, we not only outperform state-of-the-art multicore algorithms by up to two orders of magnitude, but also outperform sequential algorithms on account of good multi-threaded scalability.

A. Contributions and Outline

We study parallel skyline computation for multicore architectures with a focus on parallel scalability and raw performance. In particular, after formally introducing the problem (Section II) and before concluding (Section VIII), we:

- discuss the overlap in key principles for skyline and multicore computation, elaborating on the challenges in

†Work done while in the MADALGO group at Aarhus University.

effective skyline parallelization (Section IV);
- introduce our Hybrid algorithm based of maximizing parallel throughput in a global computation paradigm (Section V) and integrating sophisticated skyline concepts such as point-based partitioning (Section VI); and
- present extensive experiments with optimized implementations of state-of-the-art skyline algorithms (Section VII). The entire experimental suite is made public.

II. Problem Definition

In this section, we formally define the skyline. We assume a dataset $\mathcal{P}$ over d ordinal dimensions, where $p[i]$ denotes the value of $p \in \mathcal{P}$ on dimension i. The skyline is defined through the concept of dominance. We say that a point p *may dominate* another point q if it has a smaller or equal value on all dimensions (Definition 1): it is then clearly at least as good as q for any combination of attributes.[1]

Definition 1 (Potential Dominance)**.** Given $p, q \in \mathcal{P}$, p *may dominate* q, denoted $p \preceq q$, iff: $\bigwedge_{i=0}^{d-1} p[i] \leq q[i]$.

Note that $(p \preceq q \wedge q \preceq p) \implies \left(\bigwedge_{i=0}^{d-1} p[i] = q[i]\right)$, which we denote by $p \equiv q$ (i.e., p and q are coincident). We say that p *dominates* q if, additionally, p and q are not coincident (Definition 2): then there is clearly some combination of attributes for which p is a strictly better point than q.

Definition 2 (Dominance)**.** Given $p, q \in \mathcal{P}$, p *dominates* q, denoted $p \prec q$, iff: $p \preceq q \wedge q \not\preceq p$ (equiv., $p \preceq q \wedge p \not\equiv q$).

Definition 2 is equivalent to that typically presented in literature. Finally, the *skyline* (Definition 3) is the subset of points in $\mathcal{P}$ that are not dominated by any other points in $\mathcal{P}$.

Definition 3 (Skyline [4])**.** The *skyline* of dataset $\mathcal{P}$, denoted $\mathrm{SKY}(\mathcal{P})$ is the set: $\mathrm{SKY}(\mathcal{P}) = \{p \in \mathcal{P} |\ \nexists q \in \mathcal{P} : q \prec p\}$.

The objective in this paper is to compute $\mathrm{SKY}(\mathcal{P})$ efficiently using the parallelism of multicore processors.

III. Related Work

The skyline operator was introduced by Börzsönyi et al. [4] along with a basic divide-and-conquer (DnC) and a block-nested-loop (BNL) algorithm. The first improvements came from index-enabled methods, based on B-trees [20] and R-trees [18]. Subsequent literature moved from indexing to main-memory processing to support pushing the skyline operator later in the query plan after, for example, joins and selects.

The *Sort-Filter Skyline* (SFS) algorithm [9] improved the performance of BNL by incurring the preprocessing cost of sorting the data so that points are compared first to other points that are closer to the origin, since they are the most likely to prune. The LESS [11] optimizations conduct dominance tests during the sorting to reduce the input size for the BNL-style processing, and SaLSa [2] introduced a min-value sort order that makes early termination possible. All three methods have similar performance and are relatively easy to parallelize.

[1] We assume WLOG to prefer smaller values; otherwise, invert signs.

Whereas the sort-based algorithms qiuckly detect dominance relationships, the *object-based space partitioning* methods of Zhang et al. [23] and Lee and Hwang [15] detect *incomparability* quickly. Both OSP [23] and BSkyTree-P [15] recursively select a data point with which to partition the rest of the data and store the skyline points in a data structure that admits faster Phase I processing. Lee and Hwang also introduce, BSkyTree-S, which does not use recursion nor the data structure. The primary difference between OSP and BSkyTree-P is how the pivot point is selected, either as a random skyline point or as one that minimizes range, respectively.

Parallel techniques exist for the GPU [3], [8], MapReduce [17], [19], [22], and other distributed architectures (surveyed by Hose and Vlachou [12]). We focus here on multicore parallelization, for which exists three algorithms. PSkyline [13] naively cuts the dataset, processes each partition on a separate core, and then merges the results together. APSkyline [16] follows the same pattern, but uses sophisticated angle-based partitioning of the dataset that does not scale with dimensionality (the reported experiments consider $d = 5$ at most). Additionally, PSFS [13], a weaker version of our Q-Flow, was introduced in [13] as a naive baseline and VSkyline [7] modifies PSkyline to utilize SIMD instructions.

Lastly, note that the most recent thorough experimental evaluation of centralized skyline algorithms [10] pre-dated the state-of-the-art methods OSP [23] and BSkyTree [15].

IV. Key Principles for Skylines and Multicore

Here we describe key principles for obtaining efficiency in skyline (Section IV-A) and multicore (Section IV-B) algorithms, and how these principles relate (Section IV-C).

A. Skyline Computation

A *dominance test* (DT) is one check of whether $p \prec q$. Naively, the skyline can be computed by checking this relationship $n(n-1)$ times, once for each ordered pair of points. One DT can be very efficiently computed (see Section VII), but DTs are the primary operation of skyline algorithms; so, efficiency is obtained by reducing how many must be done.

DTs can be avoided in three ways:

- *transitivity*—If $p \prec q$, one can discard q immediately without compromising the result, even if $\exists r : q \prec r$, thereby avoiding any other DTs that involve q;
- *cheap filter tests*—Single values (e.g., $\mathrm{L}_1(p) = \sum p[i]$) can be precomputed for each point and can indicate $p \not\prec q$ without a full DT (e.g., $\mathrm{L}_1(p) \leq \mathrm{L}_1(q) \implies q \not\prec p$);
- *region-wise incomparability*—points can be determined incomparable based by mutual relationship to another point (e.g., $p \not\preceq s \wedge s \not\preceq p$ is evident in Figure 1b, because of their mutual relationship to the midpoint).

To illustrate region-wise incomparability, consider the points in Figure 1b, which have been assigned to 2^d partitions based on whether they are smaller or larger than the midpoint of the data space. Points in 01 (the upper left) need not be compared to points in 10 (the lower right) nor 11 (the upper right), which all have a larger x-value, which saves 4 DTs. With similar logic for 10 (the lower right), one can avoid

a further 2×3 DTs, for a total of 10 out of the possible $5 \times (5-1) = 20$.

Furthermore, *processing order* is critical. Ideally, a point is determined to be dominated early, before comparing to many other points. Then it can immediately discarded (based on transitivity), avoiding all DTs that otherwise would have subsequently been conducted. If q is compared to p before r, s, t, one saves another 3 DTs. For two random points in the same partition, each has a 0.5^d probability of dominating the other. On the other hand, a random point p in partition 01 has a 0.5 probability of dominating a random point q in partition 11, since only the y-coordinate provides uncertainty wrt dominance: p necessarily has a smaller x-coordinate than q for them to have fallen into their respective partitions. As such, it is prudent to compare points in partition 11 to those in 10 and 01 before comparing to other points in 11 because the probability of stopping early is then higher.

B. Multicore Computation

Ideal multicore computation scales elegantly with parallelism: doubling physical resources should lead to nearly a two-fold improvement in performance. Also, the overhead introduced should not be so much that a single-threaded run is uncompetitive against a natively sequential algorithm. A few key principles promote achieving this high throughput.

First, throughput requires large blocks of *independent, unordered work*. Preferably, the work is nearly identical so that the work distributed to each thread is balanced.

Second, global data structures must be either read-only or very infrequently updated during parallel processing. Otherwise, threads compete to update the data structures and either stall each other with locks or create dirty writes. Alternatively, the data structure can be updated during sequential work, but then only one thread is participating and the others idle.

Third, the choice of synchronization points must be done very carefully. One loses throughput first by waiting for the last $t-1$ threads to finish their workloads and then for the sequential work that takes place after synchronizing. Work done here is subject to Amdahl's Law: no matter of parallelism can improve beyond the cost of this phase.

Observing all these principles usually introduces some overhead relative to a natively sequential algorithm because, for example, the processing order is sub-optimal or the data structures are less current and/or complex.

C. Multicore skylines: bridging the computational principles

The principles outlined in Sections IV-A and IV-B conflict, presenting a *major* challenge for parallelization—the central role of processing order. The points in Figure 1b cannot be partitioned in any way such that both q and r can compare to points in all "lower" partitions (p and s) before being compared to each other. As soon as the computation becomes *local*, the probability of discarding a point early decreases.

The lone exception is if partition 00 is dense (e.g., when the data is highly correlated), when any random partitioning will be effective, because the dominance probability is *always* high. It will not be surprising then in Section VII when we show that state-of-the-art multicore algorithms are drastically outperformed by state-of-the-art sequential algorithms for any but low-dimensional, highly correlated data.

Furthermore, skyline points are found quite quickly, so maintaining a shared, up-to-date, sophisticated search structure over existing skyline points is challenging. The approach in [13], [16] is to not do this at all.

Algorithm 1 Q-Flow ($\mathcal{P}, \alpha \longrightarrow \mathrm{SKY}(\mathcal{P})$)

```
1:  S ← ∅.
2:  Prefilter, sort, and/or partition P.                ▷ Initialization
3:  while P ≠ ∅ do
4:      Q ← next α points of P.
5:      P ← P \ Q.
6:      for i ∈ [0, |Q|) do                             ▷ Parallel Phase I
7:          if ∃q ∈ S : q ≺ Q[i] then
8:              Mark Q[i] as pruned.
9:      Remove pruned Q[i] from Q.                      ▷ Compression
10:     for i ∈ [0, |Q|) do                             ▷ Parallel phase II
11:         if ∃j ∈ [0, i) : Q[j] ≺ Q[i] then
12:             Mark Q[i] as pruned.
13:     Remove pruned Q[i] from Q.                      ▷ Compression
14:     Append Q to S.                                  ▷ Update structure
15: return S.
```

V. Flow of Control

In this section, we introduce Q-Flow, a simplification of our proposed algorithm that illuminates the key ideas with regards to maximizing parallelism in a skyline-friendly manner.

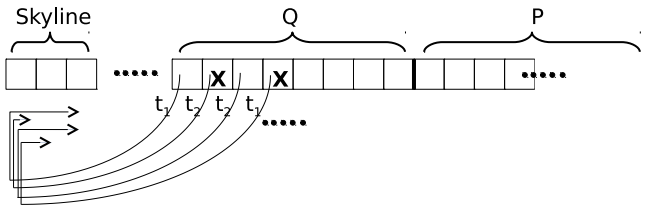
A. Overall algorithm design

Q-Flow is described in Algorithm 1. At a high-level, the points are sorted and then batch processed in blocks of size α. The sorting is chosen such that two properties are maintained: first, if p precedes q, then $q \not\prec p$. Second, points which are most likely to prune other points appear early in the sort order. In particular, Q-Flow uses Manhattan (L_1) norm.[2] Points are then processed in blocks of size α. The objective with each block is to determine whether any of the points within it are dominated. Once a block finishes, points that were not dominated are appended to the list of skyline points, since the sort order implies they will not be dominated by any as-yet-unprocessed points.

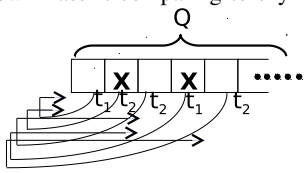
Threads are free to process points within a block out-of-order; so, nothing is guaranteed in terms of execution order within the block. However, the block is not processed until after all blocks preceding it; so, once it starts, the skyline points preceding it are completely and globally known.

The processing of a block thus occurs in two phases. First, each point p in the block is compared in parallel in the same order as a sequential algorithm to known skyline points to see if any dominate p, in which case p is flagged for removal.

[2] $L_1(p) = \sum p[i]$. One can easily verify that if $p \preceq q$, then p has a smaller or equal value on all attributes, so therefore has a smaller sum than q, too [9]. If sums are equal, then either $p \equiv q$ or $\exists i, j : p[i] < q[i] \wedge q[j] < p[j]$.



(a) Q-Flow Phase I: comparing to skyline points.



(b) Q-Flow Phase II: comparing to other survivors.

Fig. 2: The parallel phases of the Q-Flow algorithm: comparing to known skyline points (a), then other survivors (b).

Phase I shrinks the block to size $\alpha' \leq \alpha$. Phase II determines for each surviving point p whether p is dominated by any other surviving points that precede it in the block. Any points that survive both phases are therefore skylines points and appended to the global skyline list.

After each phase, we synchronize threads and remove the dominated points, which in each parallel phase had only been flagged for removal (so that neither the block, nor the dataset, nor the skyline need to be updated during parallel processing).

Overall, Q-Flow balances maximizing throughput during the expensive but parallelized phases with minimizing dominance tests using ordered computation against a global skyline.

B. Phase I: Comparing to skyline points

The most expensive phase of computation, requiring $\mathcal{O}(\alpha|S|d)$, is in comparing to known skyline points, which can be a large percentage of the input (depending on data correlation). This phase is fully parallelized. Each point p is processed by an independent thread, which iterates the known skyline points to see if any dominate p. If so, p is marked for removal and processing for p is immediately terminated. Figure 2a illustrates this process, where each point in the set Q is independently compared to each point in the known skyline. Processing for the second and fourth points terminates at the first and second skyline point, respectively, where they are first discovered to be dominated.

The early detection of dominance is enabled because we maintain a skyline shared among all threads that is accurate up to the start of the current α-block. Up to this point, comparisons to the known skyline can be done in the exact same manner as any sequential algorithm. This ensures a higher probability of detecting dominance early, minimizing DTs through transitivity.

C. Phase II: Comparing to peers

In Phase II, each point p that survived Phase I is compared to other surviving points. Processing for p needs only continue until p because of the sort order and can still abort as soon as p is dominated. This is illustrated in Figure 2b, where processing for the second and fourth points terminates at the first and third point, respectively, but the third and fifth points are compared to all preceding two and four points, respectively.

Phase II is necessarily less efficient than Phase I, because the processing is unordered. For example, in a sequential algorithm, the fifth point in Figure 2b would not be compared to the fourth point, because the fourth point is not in the skyline. However, since the first thread had not yet processed the fourth point when the second thread processed the fifth, it was not yet known that the fourth point is dominated. This phase does not exist in sequential algorithms: it is the compromise for high parallel throughput. But it is still not naive, it is fully parallelized, and it comprises only a very small percentage of overall execution time, requiring $\mathcal{O}(\alpha^2 d)$.

D. Sequential work: compression and skyline updates

Each phase is followed by a synchronization phase, which serves several purposes. Foremost, it ensures that the skyline is always correct to within α points. Secondarily, it eliminates branching and improves data locality and cache-friendliness.

After each parallel phase, we compress the α-block by shifting all the surviving points left to overwrite those that are flagged for removal, reestablishing a contiguous layout. After Phase II, where all surviving points are clearly in the skyline, we also append the compressed, contiguous block to the known skyline, all requiring $\mathcal{O}(\alpha)$ of sequential work.

Without the compression, for each newly processed point q, one would either require a DT against removed points that transitivity implies is unnecessary or a branch at every point p determining whether p has been marked for removal or not.

E. Summary of Q-Flow contributions

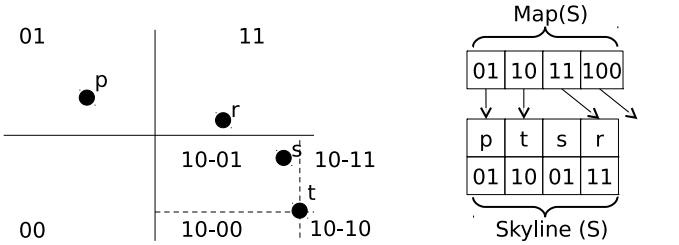
Q-Flow maintains high throughput with the majority of work in independent, parallel chunks (Phase I). Limited synchronization keeps the data contiguous and the globally known skyline within a bounded additive factor of correctness, α.

It utilizes transitivity extensively by removing dominated points as soon as a phase completes. It also utilizes cheap filter tests extensively by pre-sorting the data by L_1 norm. This eliminates half of the DTs by ensuring dominance can only occur in one direction for any given pair of points. Furthermore, it keeps the global list of skyline points ordered so that dominance can be detected earlier. Consequently, Q-Flow is already efficient. We will show in Section VII that even it outperforms the state-of-the-art multicore algorithms on moderate-to-heavy workloads.

Lines 6 and 10 are intentionally presented very abstractly. Naively, they incur quadratic loops. Our Hybrid algorithm (Section VI) minimizes the number of DTs on these lines by processing points in a more sophisticated manner, using point-based partitioning and a data structure over the known skyline.

VI. The Hybrid Multicore Skyline Algorithm

In the previous section, we introduced Q-Flow to illustrate the parallel concepts of our algorithm. In particular, Q-Flow



(a) The 2-level partitioning of skyline points for the Hybrid data structure.

(b) Physical layout of the data structure.

Fig. 3: Skyline points data structure in Hybrid.

Algorithm 2 updateS&M($S, \mathcal{M}(S), Q$)

```
Append Q to S.
Pop sentinel off of M(S).
(m, i) ← top(M(S))
for j ∈ [0, |Q|) do
    if Q[j].m= m then
        Q[j].m← part(Q[j], S[i]).
    else
        (m, i) ← (Q[j].m, |S| + j).
        Push (m, i) onto M(S).
Push sentinel (2^d, |S| + |Q|) onto M(S).
```

eliminated more than half of the DTs through its sorting by L_1 norm (a cheap filter) and its synchronization points (to facilitate transitivity by removing dominated points). Early domination was made more probable by maintaining an ordered, global skyline and processing points block-wise.

Recall from Section IV-A that there is a third method for eliminating DTs: region-wise incomparability. Here, we introduce our full algorithm, Hybrid, which uses point-based partitioning and a simple, effective data structure for the skyline points to facilitate detecting region-wise incomparability.

At a high-level, the algorithm is identical to Q-Flow, except that we first pre-filter the dataset and then partition it into 2^d-1 regions by their relationship to a (potentially virtual) *pivot point* (Section VI-A). We can then use this to enhance both Phase I (Section VI-C) and Phase II (Section VI-D) by means of a novel, simple data structure (Section VI-B).

A. Initialization and partitioning

The initialization consists of three parts: pre-filtering, partitioning, and sorting.

1) Pre-filtering: Most datasets contain points that are dominated by a relatively large percentage of the dataset, and thus are easily pruned. For correlated data, this is true of most points. Therefore, we quickly remove these points without much cost before the heavier initialization tasks such as calculating a pivot point for partitioning and sorting the data.

The filter requires two parallel passes of the dataset. In the first, each thread maintains a priority queue of the β points with smallest L_1 norm the thread has seen. Each point p is first compared to the largest of these by L_1 norm and replaces it if smaller. Otherwise, p is compared to each of the β points to see if any of them dominate it.

On the second pass, each thread grabs points and compares them to all points in all priority queues.[3]

2) Partitioning: We partition the data (after pre-filtering) based on a constant point, v, called the *pivot*. Point p is assigned to a partition with mask m, $m[i] = (p[i] < v[i] ? 0 : 1)$. The pivot thus splits the data into 2^d regions, which enables reasoning about entire regions of points at one time. In particular, two properties are worth noting. First, if $|m|$ denotes the number of bits set in a mask, then $|m| \geq |m'|, m \neq m'$ implies that no point q with mask m can dominate a point p with mask m'. This guarantee arises since q is worse relative to v on more dimensions than p is; so, clearly q is worse than p on at least one dimension.

Second, if $m \;\&\; m' < m$, then no point q with mask m can dominate a point p with mask m' because there exists a dimension on which p is better relative to v and q is worse. The masks are stored with the points and can then be used as cheap filter tests for DTs based on the above two properties.

Figure 3a illustrates how partition masks are assigned. For example, p has a smaller x-coordinate that the midpoint of the data, but a larger y-coordinate. So, it is assigned partition 01. We experiment with several methods of selecting an appropriate pivot, but typically use the median of the data, because it produces partitions of roughly equal size.

3) Sorting: Let m denote the mask assigned to points. We sort the data based on three keys: the number of bits set in the mask, $|m|$ (i.e., *the level*), the integer value of the mask, m, and finally the L_1 norm as in Q-Flow. We use a bithack to improve the efficiency of sorting by combining the level and mask as follows: We store the compound key, $\mathrm{K} = (|m| \ll d) \mid m$ (requiring $d + \lg d$ bits). Then, we can retrieve the mask $m = \mathrm{K} \;\&\; (2^d - 1)$ and the level $|m| = \mathrm{K} \gg d$. By using this composite key, we can sort by level and mask by sorting just one value.

This three-way sort has two key ideas. First, it maintains the property that $q \not\prec p$ if p precedes q in the sort order. Second, by grouping points by levels and masks, we can skip entire groups of points at once when the cheap filter tests of Section VI-A2 evaluate true.

B. Data Structure for skyline points

Because Phase I is expensive, to further minimize the required DTs, we build an auxiliary data structure $\mathcal{M}(S)$ over the known skyline points with two-level partitioning.

Figure 3 illustrates the structure, a simple vector of mask-pointer pairs. Each pair, ordered the same as the data and skyline points, contains a mask and a pointer to the first skyline point with that mask. The final pair is a sentinel indicating the end of the list. The simplicity of the data structure makes it very efficient to update (Algorithm 2), very fast to iterate, and space-efficient, because empty partitions are not stored. Since the data was originally sorted by level and mask first, and the compression phase of Section V-D always shifts data left, all partitions are contiguous; the end of a partition is the point immediately preceding the start of the next partition.

[3] $\beta = 8$ empirically configured; appreciable impact only correlated data.

Algorithm 3 compareToSky $(S, \mathcal{M}(S), q \longrightarrow \{\text{true}, \text{false}\})$

```
1: for i ∈ [0, |M(S)|) do
2:     m ← M(S)[i].m.
3:     if m is not incomparable to q.m then
4:         s ← M(S)[i].s; t ← M(S)[i + 1].s
5:         m' ← part(q, S[s]).
6:         if m' = 2^d − 1 ∧ q ≢ S[s] then return true.
7:         for j ∈ (s, t) do
8:             if S[j].m is not incomparable to m' then
9:                 if S[j] ≺ q then return true.
10: return false.
```

Algorithm 4 compareToPeers $(Q, \text{me} \longrightarrow \{\text{true}, \text{false}\})$

```
1: i ← 0.
2: while Q[i].|m| < Q[me].|m| do
3:     if Q[i] is not pruned then
4:         if Q[i].m is not incomparable to Q[me].m then
5:             if Q[i] ≺ Q[me] then return true.
6:     i ← i + 1.
7: while Q[i].m < Q[me].m do
8:     i ← i + 1.
9: while i < me do
10:    if Q[i] ≺ Q[me] then return true.
11: return false.
```

For the second level of partitioning, the pivot is simply the first point in the partition (the one with lowest L_1 norm), so that no cost is incurred in computing it. The partitioning is done on Line 6 of Algorithm 2, where a point that is not a level-2 pivot is reassigned to a new partition based on its partitions' pivot. For example, in Figure 3a, partition 10 has two points; so it repartitioned. Since t has a smaller L_1 norm, it is chosen as the pivot. Point s is assigned a new partition 01 based on its relationship to t. This is reflected in Figure 3b, where $\mathcal{M}(S)$ points to t, which retains its old mask, but s, located contiguously with t, has its new mask stored with it.

The data structure is updated on Line14 of Algorithm 1, when Q is appended to S. It is initialized with the first point, which is necessarily in the skyline, and a sentinel pointing to the second point, the position after the initially known skyline.

C. Phase I: comparing to skyline

For Q-Flow, Phase I consisted of a naive quadratic loop. For Hybrid's Phase I, described in Algorithm 3, we take advantage of partitioning twice, using $\mathcal{M}(S)$. When processing point q, rather than iterating all points $p \in S$, we iterate all masks in $\mathcal{M}(S)$ (Line 1). For each mask, we first conduct a cheap filter test to see if points with that mask can dominate points with q's mask. If not, we skip it, and all the corresponding DTs, entirely. Otherwise, we jump straight to that partition in the skyline list S (Line 4) and process those points contiguously.

Second, we compare q to the pivot point v of the partition (pointed to by $\mathcal{M}(S)$) to produce a new mask m' (Line 5). Since all points in the partition have been reassigned new masks based on v, we can use m' as a stronger cheap filter test for the rest of the points in the partition (Line 8): if the second property of Section VI-A2 holds, we can skip the DT.

Consequently, we are able to derive most of the benefit of recursive point-based partitioning [15], [23] in terms of skipping DTs in Phase I without the need for either recursion nor complex tree structures. Furthermore, we vastly reduce the number of DTs required in Phase I relative to Q-Flow.

D. Phase II: comparing to peers

In Phase II, a point q is compared to points for which there is no data structure $\mathcal{M}(S)$, which is updated *after* Phase II. It is not yet known whether any (except the first) of these points are skyline points. Although we cannot take advantage of the data structure, we can still exploit the partitioning, because data is sorted by partition. Algorithm 4 gives Phase II of Hybrid.

Each point q is compared to those preceding it in the same α-block. We can decompose the iteration over those points into three consecutive loops, each maintaining different invariants. The first points are those in levels less than that of q (Line 2). For these points, it is worthwhile to conduct a cheap filter test based on masks to detect region-wise incomparability (Line 4). The second set of points all have the same level as q but are in different partitions (Line 7). All of these, by the first property of Section VI-A2, are necessarily incomparable to q and can be skipped. Finally, the remaining points that precede q are in the same partition (Line 9). In this case, no assumption can be made, and the DT must be conducted (Line 10).

E. Summary of Hybrid contributions

In summary, Hybrid benefits from the high throughput flow of control in Q-Flow *and* state-of-the-art point-based partitioning ideas. It maintains a global, shared skyline with an efficient, simple data structure that dramatically reduces the number of DTs in Phase I. Phase II also benefits from the level-mask-based sort order. The for loop can be decomposed into three loops, each maintaining different assumptions about the data. In the next section we show that, indeed, we substantially reduce DTs, yielding state-of-the-art performance.

VII. Experimental Evaluation

In this section, we conduct a thorough empirical evaluation of the run-time performance and scalability of our proposed algorithm, relative to the state-of-the-art alternatives in literature [13], [15]. This includes our custom parallelization of the sequential BSkyTree algorithm, which we call PBSkyTree. Furthermore, we extensively contrast Hybrid and Q-Flow in order to characterize the impact of our data structure and partitioning, and we investigate the impact of the block-size, α, and selection of pivots, v.

A. Experimental Setup

1) Hardware: We run all experiments on the same machine with 64 GB of RAM and a dual eight-core Xeon E5-2670 processor clocked at 2.67 GHz. The machine has Hyper-

Threading disabled so we experiment with at most 16 threads.[4] All skyline methods are executed completely in main memory.

2) Software: We implement both Q-Flow (Section V) and Hybrid (Section VI). Additionally, we consider three state-of-the-art alternatives:

PSkyline [13]: the state-of-the-art multicore algorithm based on a simple divide-and-conquer strategy where a dataset is linearly partitioned into smaller blocks—one per thread—of equal size. Each thread computes a local skyline within its block (aka Phase I) and then the resulting skyline set is merged into a global skyline (aka Phase II). Both phases are efficiently parallelized specifically to exploit multicore architectures.

BSkyTree [15]: the state-of-the-art sequential algorithm that employs sophisticated pivot selection and recursive point-based space partitioning techniques to significantly reduce DTs. We use its most efficient variant, BSkyTree-P, shown to outperform all other sequential algorithms [15].

PBSkyTree: our parallelized version of BSkyTree. It is not trivial to parallelize BSkyTree because it uses a dynamic tree structure (SkyTree) to store skyline points, which is constantly updated, and the algorithm is fully recursive. Details of parallelizing BSkyTree are in Appendix A.

In general, significant improvements can come from vectorization of dominance tests (DTs) [7]. We extend the techniques of 4-degree data-level parallelism (128-bit SIMD registers and SSE) in [7] to obtain 8-degree data-level parallelism supported by our experimental hardware (256-bit SIMD registers and AVX). The vectorized DTs are used by all algorithms for speedups of $1.75\times$, $1.32\times$, $2\times$, and $1.25\times$ in PSkyline, BSkyTree, Q-Flow, and Hybrid, respectively (under our default workload, $n = 1$M and $d = 12$).

Implementations are written in C++ using a common interface and the same optimized dominance tests for a fair comparison. The CPU code is compiled using `g++` (v4.4.7) with the `-O3` optimization flag. For multi-threading, we use the OpenMP API (v3.0). We publicly release the extensible collection of algorithms, called *SkyBench*.[5]

3) Datasets: We generate correlated, independent, and anti-correlated synthetic datasets using the standard skyline data generator from [4]. We adopt (and extend) the parametres from the existing skyline research so that results can be directly compared. In particular, we vary dimensionality, d, from 4 to 16 and cardinality, n, from 500K to 8M. Figure 4 shows the corresponding skyline sizes for all three distributions. By default, as in [15], we set $d = 12$ and $n = 1$M.

We also include three real datasets. NBA and House are described in [15] and Weather is described in [6]; although, we use more dimensions and years of data than [6].[6] Statistics for the real datasets are given in Table I.

[4]Experiments with 8 threads on our less powerful but hyperthreading-enabled 4-core machine show that hyper-threading improves Hybrid by 29% and PBSkyTree by 38% under the default, independent workload ($d = 12$, $n = 10^6$). Using 16 threads slows performance for both algorithms. Thus there is some limited potential parallelism for hyperthreads to exploit too.

[5]http://cs.au.dk/research/research-areas/data-intensive-systems/repository/

[6]Preprocessing scripts for datasets are also available within *SkyBench*.

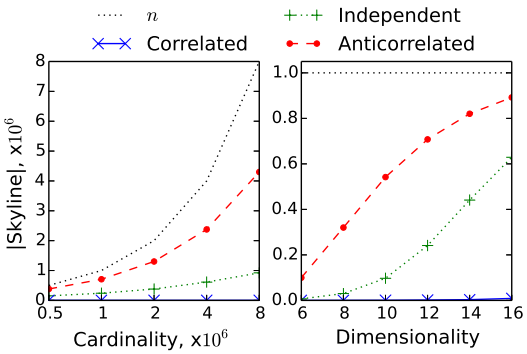


Fig. 4: Skyline sizes in synthetic data.

Dataset	Cardinality	Dimensionality	|SKY|
NBA	17,264	8	1,796 (10.40%)
HOUSE	127,931	6	5,774 (4.51%)
WEATHER	566,268	15	63,398 (11.20%)

TABLE I: Specifications of real datasets.

B. Overall Performance of Algorithms

We begin by comparing the performance of Hybrid to the state-of-the-art sequential and parallel algorithms. We also include Q-Flow and our parallel version of BSkyTree. The four parallel algorithms are run with 16 threads ($t = 16$) to utilize all available cores.

1) Varying Dimensionality: Figure 5 reports running times for the five algorithms on each distribution as a function of d. Naturally, irrespective of distribution, all algorithms slow with increases in d because the skyline size grows (c.f., Figure 4).

Consider first the primary multicore algorithms, PBSkyTree, Hybrid, and PSkyline on the correlated data (top-right subfigure). While Hybrid is outperformed by PSkyline when $d \leq 12$ and by PBSkyTree when $d > 12$, all algorithms are competitive and finish within 70 ms at 12 dimensions and within 300 ms at 16 dimensions. On the more challenging workloads, the independent (bottom-left) and anticorrelated (bottom-right) data, Hybrid is the clear best performer across all dimensions. Due to its carefully crafted partitioning scheme, Hybrid is able to skip most of the DTs for points within incomparable regions, a figure that grows with d. PSkyline, by contrast, has no means to recognize this incomparability and thus suffers heavily–it is the worst performing algorithm of all five. PBSkyTree, on the other hand, cannot maintain the high throughput of Hybrid; so, the performance diverges as the increased workloads provide more opportunity for parallelism. Overall, Hybrid scales the best with increasing d among all skyline methods and achieves one order of magnitude better performance by $d = 16$.

Next, consider Q-Flow, the simplification of Hybrid. We can see that in all cases it is substantially slower than Hybrid, indicating that the use of region-wise incomparability and our partition data structure pays significant dividends. Furthermore, except on correlated data, Q-Flow is $2\times$ faster than PSkyline on average. This illustrates that optimizing throughput is enough already to outperform the state-of-the-

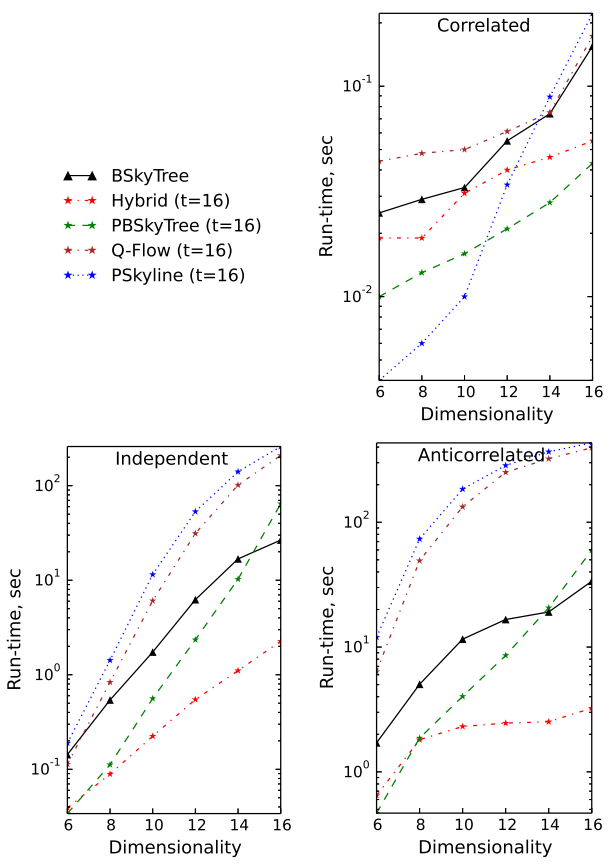


Fig. 5: State-of-the-art performance w.r.t. d ($n = 1$M).

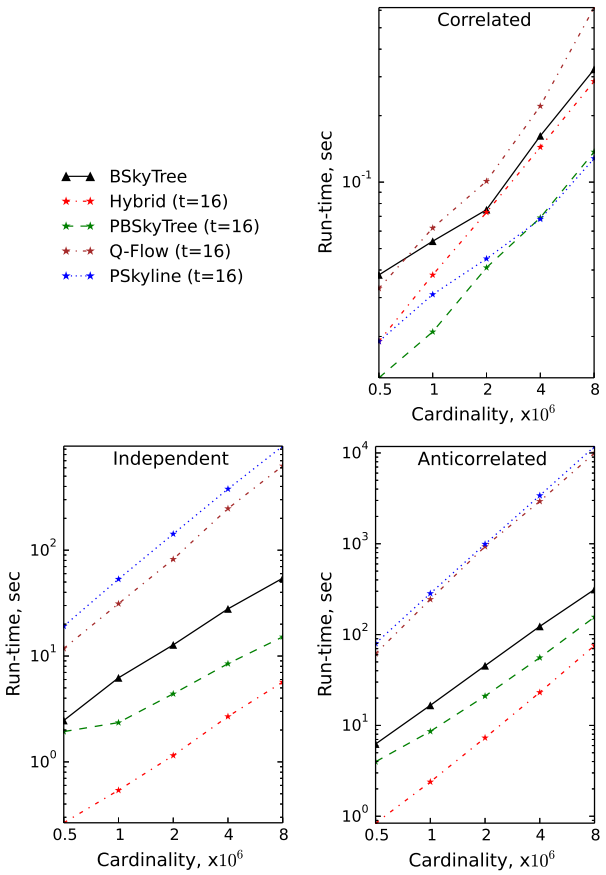


Fig. 6: State-of-the-art performance w.r.t. n ($d = 12$).

art multicore algorithm, highlighting the importance thereof. Still, both Q-Flow and PSkyline, without advanced techniques for reducing expensive DTs, are outperformed even by single-threaded BSkyTree by up to an order of magnitude. Our parallelized PBSkyTree further achieves up to $5\times$ speedup over BSkyTree; so, on independent and anticorrelated data, the region-wise incomparability is critical for competitiveness.

Note that PBSkyTree is unable to make use of its techniques to the same extent and is worse than sequential BSkyTree with $d > 14$. This is because threads idle or receive imbalanced workloads when the partitions become too small; so, the overhead of thread management is never amortized.

Notably, none of the advanced techniques pay off on the low-dimensional correlated data, where one finds a tiny skyline (c.f., Figure 4). This is because most points are eliminated on their first DT. As such, PSkyline with no initialization overhead (trivial division of points to threads) compared to other methods achieves the best performance. However, with increasing d, it eventually becomes the worst performer ceding the place to our parallelized PBSkyTree, because the skyline size grows closer to that of the other distributions.

2) Varying Cardinality: Figure 6 instead shows the effect of increasing cardinality. Here, like with d, increases naturally slow all algorithms, because they increase the number of pairs of points that must be compared. Again, Hybrid reports the fastest times on independent and anticorrelated data outperforming the closest competitor, our parallelized PBSkyTree, by a multiplicative factor of 2 to 7. All correlated workloads are completed by all algorithms in less than 50 milliseconds.

Algorithm	NBA		House		Weather	
	msec.	speedup	msec.	speedup	sec.	speedup
BSkyTree	12	–	42	–	6.50	–
PBSkyTree	6	2×	18	3×	1.44	8.5×
PSkyline	9	2.8×	54	5.3×	3.61	8.7×
Q-Flow	4	8.2×	30	9×	6.18	11.6×
Hybrid	4	5.7×	12	7.2×	0.89	13×

TABLE II: Performance on real data.

The relative gap between all algorithms excluding PBSkyTree remains constant with increasing n. The relative improvement for PBSkyTree is explained conversely to its degradation with respect to d: a larger input size creates larger partitions. Consequently, it can better utilize the physical resources than it can on smaller datasets.

3) Real Datasets: A main reason to evaluate skyline algorithms on real datasets is to verify their efficiency when distinct value condition cannot be assumed (i.e., values can be duplicated in the dataset). Table II shows the corresponding run-times with $t = 16$ and speedups over $t = 1$.

Since NBA and House datasets are relatively small, we do not obtain significant performance gains with multi-threaded algorithms. Though, Q-Flow is at least $8.2\times$ faster than its single-threaded run. This is due to efficient use of threads in the

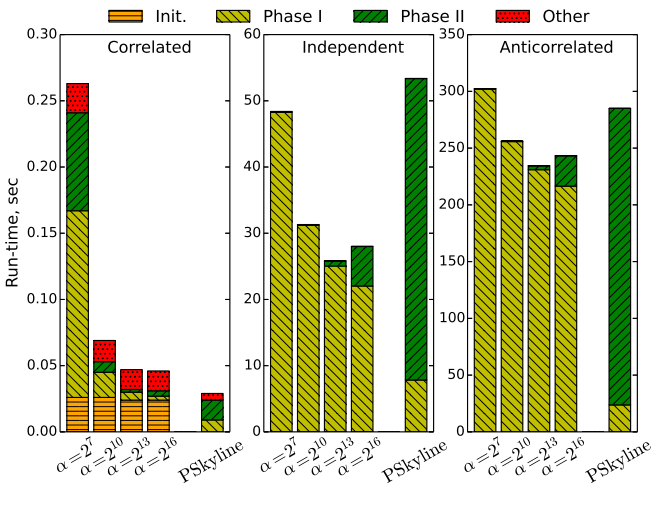


Fig. 7: Effect of α in Q-Flow with varied α ($n = 1$M, $d = 12$)

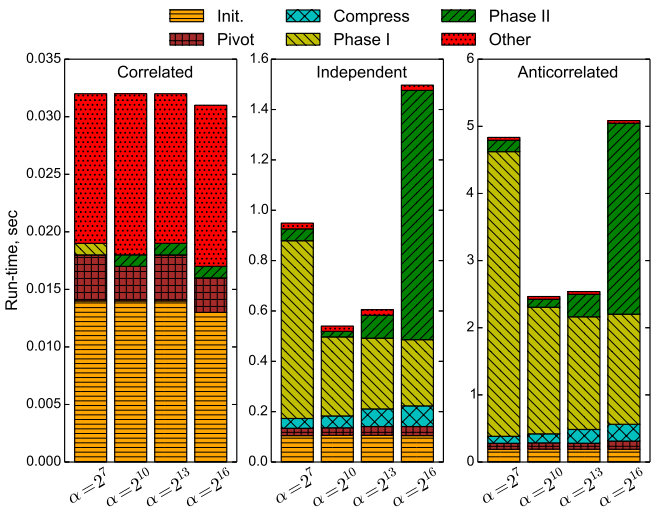


Fig. 8: Effect of α on Hybrid with varied α ($n = 1$M, $d = 12$)

initialization phase (computing L_1 for each point in parallel). Hybrid has a pre-filtering phase that is not amortized; so, the speedup is smaller.

On Weather, the most challenging real dataset, we observe $\geq 8.5\times$ speedup by all parallel algorithms; they all outperform single-threaded BSkyTree. Hybrid achieves almost linear scalability with a $13\times$ speedup (of the theoretical limit, $16\times$).

Our main observation is that while Q-Flow is outperformed in most cases on real datasets, highly optimized Hybrid is the best performer without exception. Its 16-threaded runs benefit most from each core and outperform all other counterparts.

The results here are not surprising, since the absence of distinct value condition effects mostly the efficiency of the DTs (and to some extent, the size of the skyline). However, our optimized and vectorized DTs mitigates this very effectively.

C. Refined performance analysis

Having established in the previous section the overall performance characteristics of the algorithms, we now turn to a more granular analysis of Q-Flow, PSkyline, and Hybrid to elucidate better explanations for their relative performance. We first look at the effect of α (Section VII-C1). We report these results with the running times decomposed into the various phases of the algorithm. Then, we look at the effect of the pivot selection (Section VII-C2).

1) Granular performance study with effect of α: As described in Section V, Q-Flow processes points in α-sized blocks. To evaluate its impact, we vary α from 128 to 64K and report the results in Figure 7. We measure the execution time and break it down into the phases of Q-Flow: compute L_1 and sort ("Init."), Phase I, Phase II, and the rest ("Other"). For comparison, we also show the execution time of PSkyline. In PSkyline, there is no initialization phase, while Phase I and Phase II correspond to local (parallel map) and global (parallel merge) skyline computation, respectively.

We make the following three observations. First, setting α optimally results in up to $5.7\times$, $1.9\times$, and $1.3\times$ speedups on correlated, independent, and anticorrelated data, respectively. Notably, the same α value ($\alpha = 2^{13}$) is optimal under all three distributions and we therefore use (and have used) it in all other experiments. Second, the initialization overhead (≈ 25 ms) is relevant only on correlated data, where the resulting skyline is tiny and thus computed very fast (in a few ms). While α generally helps to balance the computation between the phases, Phase I still dominates the entire processing on independent and anticorrelated distributions. Third, our simplified algorithm, Q-Flow, outperforms the state-of-the-art multicore PSkyline algorithm on all but correlated data. This is because all local skyline computations are isolated and cannot benefit from the global skyline until the merge phase starts. As such, Phase II is the most expensive part in PSkyline. On correlated data, by the time Q-Flow computes L_1 for each point and sorts the data, PSkyline computes the actual skyline.

Also note the relative time spent by each algorithm in Phase I/II, a result of the different flows of control. This implies that the multi-threaded scalability of the algorithms is dependent on parallelizing different computation. For Q-Flow, it is critical to effectively parallelize Phase I; for PSkyline, Phase II.

Similarly, we evaluate the effect α has on Hybrid in Figure 8. Since Hybrid involves more computational categories, we decompose time into additional groups. The extra timings include prefiltering using priority queues ("Pre-filter"), pivot selection ("Pivot"), and α-block compression ("Compress").

For Hybrid, the precise setting of α has less impact on performance, offering at best a speedup of $2\times$ (relative to $\alpha = 2^7$). The ideal value is again consistent, although this time slightly smaller, $\alpha = 2^{10}$. Regarding performance, note that the scales in Figure 8 differ from those in Figure 7, and that Hybrid outperforms both other algorithms (as we saw in the previous subsection).

The decomposed running times are revealing. On the correlated data, virtually no time is spent in Phase I nor Phase II: the pre-filtering consumes half the cost, but nearly produces the solution. The total time is doubled by the 15ms of overhead involved in other tasks such as constructing the data structure.

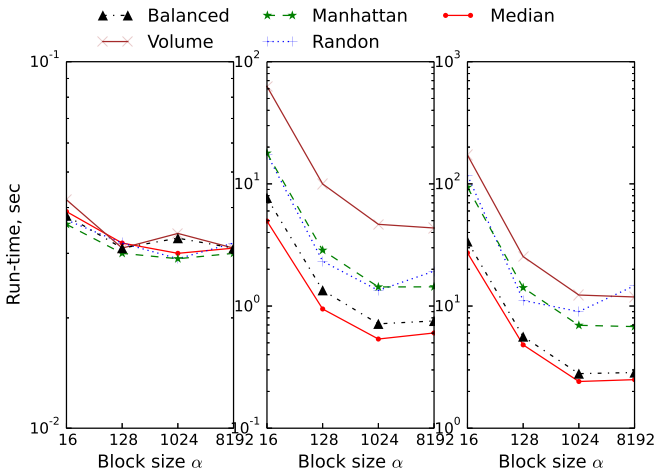


Fig. 9: Effect of pivot selection in Hybrid ($n = 1$M, $d = 12$)

On independent and anticorrelated data, the initialization and other costs shrink proportionately as the skyline size grows and more DTs are required. In both cases, the fully parallelized Phase I becomes the dominant cost, as with Q-Flow. This is ideal, because Phase I contains the most sophisticated DT-reduction techniques and is the most scalable with additional hardware resources. Together, Phase I and Phase II, the parallel components, combine for up to 95% of computation.

2) Impact of pivot selection in Hybrid*:* Next, recall from Section VI-A2 that Hybrid partitions data based on a selected pivot, v. Here, we investigate five methods of selecting v:

- **Balanced** [15]: the skyline point with minimum range;
- **Manhattan** [9]: the point with minimum L_1 norm, which is necessarily a skyline point[7];
- **Median**: virtual point where each dimension is the median of points surviving pre-filtering;
- **Volume** [2]: the point with maximum $\Pi_i p[i]$, which is necessarily a skyline point;
- **Random** [23]: A (non-uniform) random skyline point.[8]

Figure 9 shows the effect of the pivot selection methods relative to α for correlated, independent, and anti-correlated default workloads, respectively. On correlated data, the selection method has limited impact—all methods perform more-orless equally—because there is so little Phase I and Phase II processing to be done. On the other workloads, Median performs consistently best, and Balanced is a clear second-best. Both techniques provide well balanced partition sizes, which is ideal for skipping large blocks of points with region-wise incomparability. All methods demonstrate the same trend with respect to α as in Figure 8, confirming that these trends are independent of pivot selection method.

D. Multi-threaded Parallel Scalability

Lastly but importantly, we report the multi-threaded scalability of the parallel algorithms. In Figures 10 and 11, we show how Q-Flow and PSkyline scale with increasing number of threads when varying dimensionality and cardinality, respectively. Both algorithms scale linearly with increasing number of threads and Q-Flow thus preserves its advantages on all but less challenging datasets. In Figure 10, Q-Flow is up to $2\times$ faster than PSkyline on anticorrelated data with all d and on correlated and independent data with $d \geq 14$ and $d \geq 6$, respectively. In Figure 11, Q-Flow is up to $1.7\times$ and $1.3\times$ faster on independent and anticorrelated data. On correlated data, due to the initialization overhead, which is $\mathcal{O}(n)$, Q-Flow is up to $4\times$ slower. Nevertheless, skyline computation of correlated data with $n = 8$M ($d = 12$) takes less than 1s with more than two threads ($t > 2$) in Q-Flow.

In Figures 12 and 13, we show how Hybrid scales with increasing number of threads when varying dimensionality and cardinality, respectively. For comparison, we also include the scalability of our parallelized PBSkyTree.

Both Hybrid and PBSkyTree scale linearly with increasing number of threads and thus benefit from each core in a system. This confirms our fair implementation of PBSkyTree. With respect to d (Figure 12), Hybrid scales significantly better on independent and anticorrelated data, outperforming PBSkyTree by more than an order of magnitude with $d = 16$. The performance difference is especially substantial with $d > 10$. This is because the progressively smaller partition sizes hurt the throughput of PBSkyTree, whereas Hybrid, based on Q-Flow maintains the high throughput. As before, Hybrid is outperformed due to its initialization overhead on easy, correlated workloads, where the run-times are below 100 ms for $t > 2$.

With respect to n (Figure 13), the performance decreases linearly by both algorithms. However, 4 and 8 threads are enough for Hybrid to outperform a 16-threaded run of PBSkyTree on independent and anticorrelated data, respectively. Again, on correlated data, due to the initialization overhead (inherited from Q-Flow), Hybrid falls behind PBSkyTree as well. However, the run-times are much faster and all runs finish within half of a second.

The overall strong multi-threaded scalability of Q-Flow and Hybrid arises from the dominance of the sophisticated, fully parallelized Phase I in the computational cost that was observed in Figures 7 and 8.

VIII. Conclusion

In this paper, we investigated scalable and efficient parallelization of skyline algorithms on multicore architectures. We first introduced a simplification, Q-Flow, of our algorithm to demonstate how we achieve high parallel throughput. We then introduced our full algorithm, Hybrid, which is the first multicore skyline algorithm to maintain a global, shared skyline that can be used to minimize dominance tests. It is also the first multicore skyline algorithm to exploit ideas of object-based partitioning. In an extensive experimental study, we show that even Q-Flow, which also maintains a global, shared skyline, outperforms state-of-the-art multicore skyline computation, and that Hybrid outperforms both the most efficient skyline algorithm in literature and our non-trivial parallelization of it. Our decomposition of performance reveals that on non-correlated data Hybrid performs well on account

[7]Our level 2 partitioning in the data structure always uses Manhattan

[8]For one pass, we select a uniform random point, v, then iterate the dataset conducting one-way DTs and replacing v whenever it is dominated

of primarily spending time in the most sophisticated and best parallelized phase of computation.

ACKNOWLEDGMENTS

This research was supported in part by the Danish Council for Strategic Research, grant 10-092316, and by the Danish National Research Foundation, grant DNRF84, for the Center for Massive Data Algorithmics (MADALGO). We thank the authors of BSkyTree [15] and PSkyline [13] for their implementations off which we based our implementations.

REFERENCES

[1] M. Alrifai, D. Skoutas, and T. Risse, "Selecting skyline services for qos-based web service composition," in *Proc. WWW*, 2010, pp. 11–20.

[2] I. Bartolini, P. Ciaccia, and M. Patella, "Efficient sort-based skyline evaluation," *TODS*, vol. 33, no. 4, pp. 31:1–49, 2008.

[3] K. S. Bøgh, I. Assent, and M. Magnani, "Efficient gpu-based skyline computation," in *Proc. DaMoN*, 2013, pp. 5:1–6.

[4] S. Börzsönyi, D. Kossman, and K. Stocker, "The skyline operator," in *Proc. ICDE*, 2001, pp. 421–430.

[5] S. Chester, M. L. Mortensen, and I. Assent, "On the suitability of skyline queries for data exploration," in *Proc. ExploreDB*, 2014, pp. 161–166.

[6] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides, "Computing k-regret minimizing sets," *PVLDB*, vol. 7, no. 5, pp. 389–400, 2014.

[7] S.-R. Cho, J. Lee, S.-W. Hwang, H. Han, and S.-W. Lee, "VSkyline: Vectorization for efficient skyline computation," *SIGMOD Rec.*, vol. 39, no. 2, pp. 19–26, 2010.

[8] W. Choi, L. Liu, and B. Yu, "Multi-criteria decision making with skyline computation," in *Proc. IRI*, 2012, pp. 316–323.

[9] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *Proc. ICDE*, 2003, pp. 717–719.

[10] H. Eder and F. Wei, "Evaluation of skyline algorithms in postgresql," in *Proc. IDEAS*, 2009, pp. 334–337.

[11] P. Godfrey, R. Shipley, and J. Gryz, "Algorithms and analyses for maximal vector computation," *VLDB J*, vol. 16, no. 1, pp. 5–28, 2007.

[12] K. Hose and A. Vlachou, "A survey of skyline processing in highly distributed environments," *VLDBJ*, vol. 21, no. 3, pp. 359–384, 2012.

[13] H. Im, J. Park, and S. Park, "Parallel skyline computation on multicore architectures," *Inf. Syst.*, vol. 36, no. 4, pp. 808–823, 2011.

[14] H.-P. Kriegel, M. Renz, and M. Schubert, "Route skyline queries: a multi-preference path planning approach," in *Proc. ICDE*, 2010, pp. 261–272.

[15] J. Lee and S.-w. Hwang, "Scalable skyline computation using a balanced pivot selection technique," *Inf. Syst.*, vol. 39, pp. 1–24, 2014.

[16] S. Liknes, A. Vlachou, C. Doulkeridis, and K. Nørvåg, "APSkyline: Improved skyline computation for multicore architectures," in *Proc. DASFAA*, 2014, pp. 312–326.

[17] K. Mullesgaard, J. L. Pederseny, H. Lu, and Y. Zhou, "Efficient skyline computation in MapReduce," in *Proc. EDBT*, 2014, pp. 37–48.

[18] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *TODS*, vol. 30, no. 1, pp. 41–82, 2005.

[19] Y. Park, J.-K. Min, and K. Shim, "Parallel computation of skyline and reverse skyline queries using mapreduce," *PVLDB*, vol. 6, no. 14, pp. 2002–2013, 2013.

[20] K.-L. Tan, P.-K. Eng, and B. C. Ooi, "Efficient progressive skyline computation," in *Proc. VLDB*, 2001, pp. 301–310.

[21] B. Yang, C. Guo, C. S. Jensen, M. Kaul, and S. Shang, "Stochastic skyline route planning under time-varying uncertainty," in *Proc. ICDE*, 2014, pp. 136–147.

[22] B. Zhang, S. Zhou, and J. Guan, "Adapting skyline computation to the mapreduce framework: Algorithms and experiments," in *Proc. DASFAA*, 2011, pp. 403–414.

[23] S. Zhang, N. Mamoulis, and D. W. Cheung, "Scalable skyline computation using object-based space partitioning," in *Proc. SIGMOD*, 2009, pp. 483–494.

Distribution	$n = 500$K	$n = 1$M	$n = 2$M	$n = 4$M	$n = 8$M
Correlated	$1.0\times$	$1.3\times$	$1.9\times$	$1.4\times$	$1.4\times$
Independent	$3.9\times$	$3.2\times$	$3.3\times$	$3.2\times$	$3.3\times$
Anti-correlated	$4.9\times$	$5.0\times$	$5.2\times$	$5.6\times$	$6.7\times$

TABLE III: BSkyTree relative to PBSkyTree ($d = 12$, $t = 1$)

APPENDIX

A. Parallelization of BSkyTree

A central idea of BSkyTree [15] is a depth-first recursion, which ensures that a point p is compared *only* to partitions that partially dominate the partition of p, and *only after* those partitions have been completely processed. Consequently, p requires far fewer dominance tests. So, the primary challenge with parallelizing BSkyTree is to parallelize the depth-first recursion. Launching threads early in the recursion sacrifices control over the processing order; threads launched late in the recursion when partitions are small have high relative overhead given that the time required for only a few points to traverse a SkyTree is neglible. Figures 5 and 6 illustrate this effect: with increasing dimensionality, the partitions become smaller (2^d increases but n is fixed) and the performance of the sequential and parallel algorithms converge. With increasing cardinality (2^d is fixed but n increases), the partitions become larger and the performance diverges.

We use two ideas to create PBSkyTree. First, we halt the recursion when there are fewer than 64 points on which to recurse; recursing futher only adds overhead. (This number was determined experimentally.) Second, at the base of the recursion, a leaf of the recursion tree is being processed and its right siblings are queued. Each of the right siblings represents a partition that has not yet been sub-partitioned (and may itself be a leaf). To increase the amount of parallel work, we group together the current node and its leftmost right siblings to create a large work batch of up to (16 * num_threads_) points to process in parallel. (This number was also tuned experimentally.) Thus, we still recurse deeply, taking advantage of the extra information of region-wise incomparability created on each recursive call, yet accumulate large blocks of parallel work. Each point in the work batch is then independently compared to all the previous points, including the partial dominance (Phase I: comparing many points to the current skyline) that comprises the majority of work in the sequential algorithm. We do not parallelize the pivot selection: it incurs negligible cost. We parallelize partitioning as in Hybrid.

Table III indicates the overhead introduced into BSkyTree by our parallelization. We execute PBSkyTree with 1 thread and compare against the natively sequential BSkyTree. We expect overhead because the last point in a work batch is potentially processed (16 * num_threads_) points too early (relative to sequential), in which case there are up to (16 * num_threads_) points to which it might not have been compared at all in the sequential version. Nonetheless, for $d = 12$, the overhead can be absorbed by 2-8 threads, depending on the dataset.

Recall for contrast that Hybrid does not process parallel batches of work along partition boundaries, instead selecting a constant work batch size, α. This nicely circumvents the difficulties in parallelizing BSkyTree.

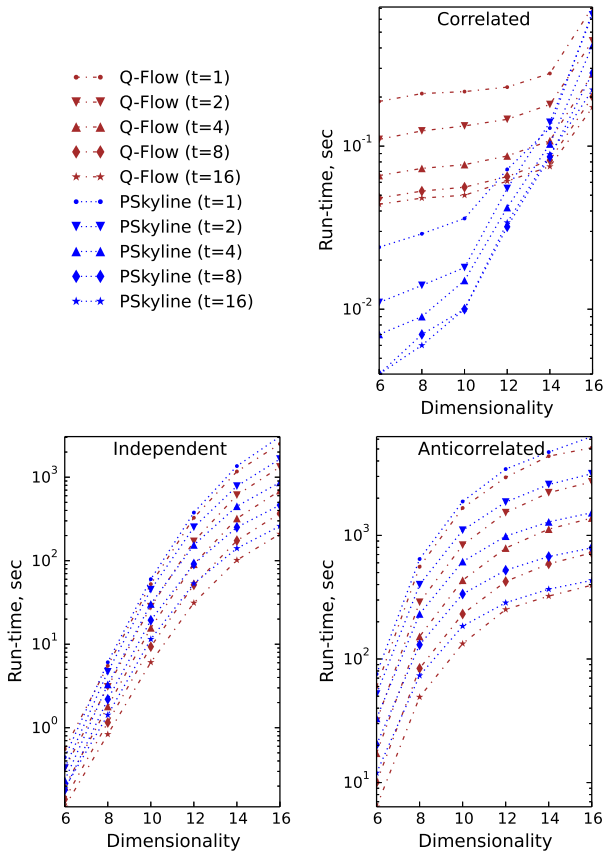


Fig. 10: Q-Flow versus PSkyline w.r.t. d ($n = 1$M).

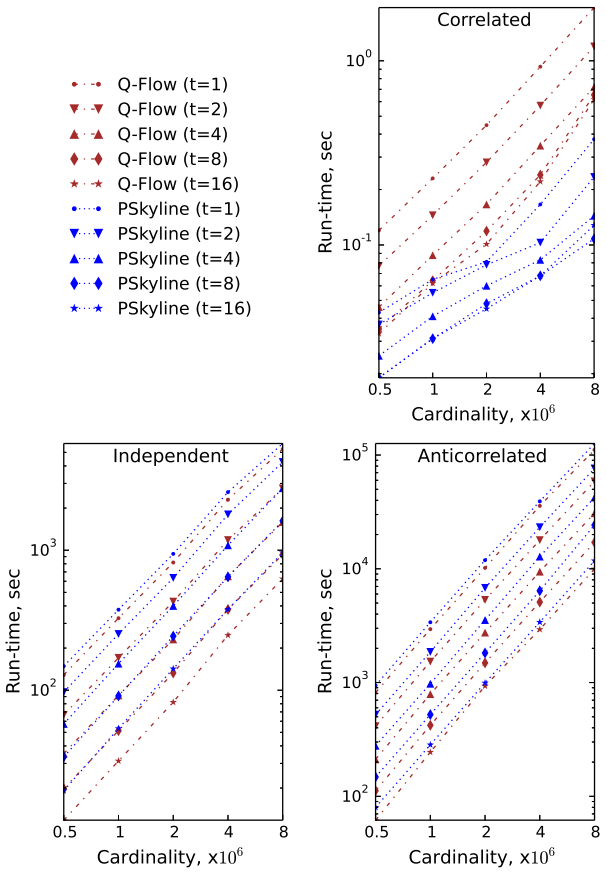


Fig. 11: Q-Flow versus PSkyline w.r.t. n ($d = 12$).

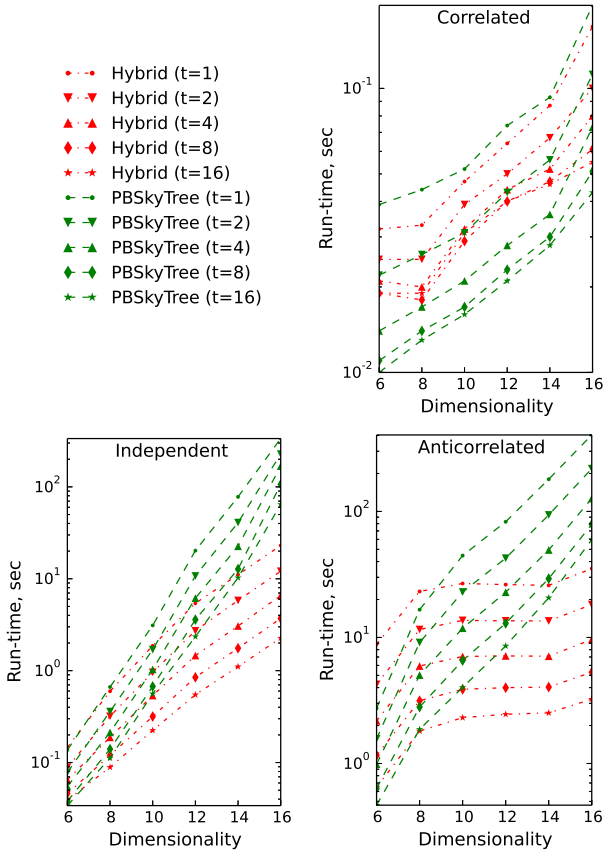


Fig. 12: Parallel Scalability in Hybrid w.r.t. d ($n = 1$M).

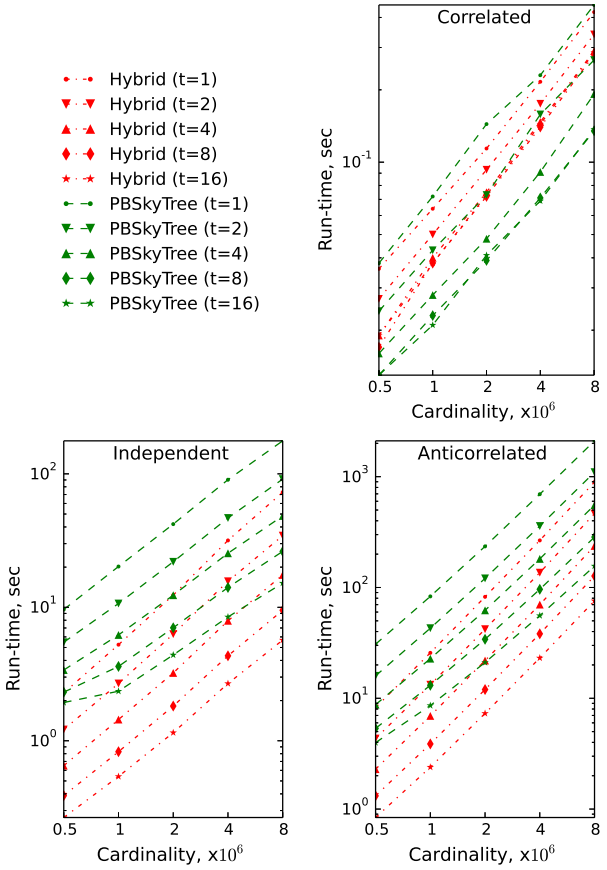


Fig. 13: Parallel Scalability in Hybrid w.r.t. n ($d = 12$).