APSkyline: Improved Skyline Computation for Multicore Architectures

Stian Liknes[1], Akrivi Vlachou[1,2], Christos Doulkeridis[3], and Kjetil Nørvåg[1]

[1] Norwegian University of Science and Technology (NTNU), Trondheim, Norway
[2] Institute for the Management of Information Systems, R.C. "Athena", Athens, Greece
[3] Department of Digital Systems, University of Piraeus, Greece
stianlik@gmail.com, {noervaag,vlachou,cdoulk}@idi.ntnu.no

Abstract. The trend towards in-memory analytics and CPUs with an increasing number of cores calls for new algorithms that can efficiently utilize the available resources. This need is particularly evident in the case of CPU-intensive query operators. One example of such a query with applicability in data analytics is the skyline query. In this paper, we present APSkyline, a new approach for multicore skyline query processing, which adheres to the partition-execute-merge framework. Contrary to existing research, we focus on the partitioning phase to achieve significant performance gains, an issue largely overlooked in previous work in multicore processing. In particular, APSkyline employs an angle-based partitioning approach, which increases the degree of pruning that can be achieved in the execute phase, thus significantly reducing the number of candidate points that need to be checked in the final merging phase. APSkyline is extremely efficient for hard cases of skyline processing, as in the cases of datasets with large skyline result sets, where it is meaningful to exploit multicore processing.

1 Introduction

The trend towards in-memory analytics and CPUs with an increasing number of cores calls for new algorithms that can efficiently utilize the available resources. This need is particularly evident in the case of CPU-intensive query operators. One example of such a query with applicability in data analytics is the skyline query. Given a set P of multidimensional data points in a d-dimensional space D, the skyline query retrieves all points that are not dominated by any other point. A point $p \in P$ is said to *dominate* another point $q \in P$, denoted as $p \prec q$, if (1) on every dimension $d_i \in D$, $p_i \leq q_i$; and (2) on at least one dimension $d_j \in D$, $p_j < q_j$. The *skyline* is a set of points $SKY_P \subseteq P$ which are not dominated by any other point in P. Computing the skyline query in a multicore context is an interesting research problem that has not been sufficiently studied yet.

Previous approaches to solve this problem are variants of the *partition-execute-merge* framework where the dataset is split into N partitions (one for each core), then the local skyline set for each partition is computed, and finally the skyline set is determined by merging these local skyline sets. However, the existing works largely overlook the important phase of partitioning, and naively apply a plain random partitioning of input data to cores. As also observed in [2], a naive partitioning approach may work well

S.S. Bhowmick et al. (Eds.): DASFAA 2014, Part I, LNCS 8421, pp. 312–326, 2014.

in case of cheap query operators on multicore architectures, while sophisticated partitioning methods may not be beneficial. Nevertheless, in the case of skyline queries which are CPU-intensive and especially for the case of "hard" datasets having an anti-correlated distribution, which are frequent in real applications [13] and also more costly to compute, the role of partitioning is significant. Skyline queries over anti-correlated data produce a skyline set of high cardinality and a naive partitioning produces local skyline sets containing uneven portions of the final skyline set and many local skyline points not part of the skyline set. In consequence, the merging cost is significantly higher than necessary, and the final result is prohibitively long processing times.

Motivated by this observation, in this paper, we present *APSkyline*, an approach for efficient multicore computation of skyline sets. In APSkyline, we employ a more sophisticated partitioning technique that relies on angle-based partitioning [15]. Since multicore skyline processing differs from skyline processing in other parallel environments, we apply all necessary adaptations of the partitioning technique for a multicore system, which combined with additional optimizations result in significant performance gains. Even though our partitioning entails extra processing cost, the degree of pruning that can be achieved in the execute phase incurs significant savings in the subsequent phases, thus reducing the overall execution time considerably. Furthermore, we show how the partitioning task can be parallelized and efficiently utilize all cores for this problem. We provide an extensive evaluation that compares APSkyline with the state-of-the-art algorithms for multicore skyline processing, which demonstrates that our approach outperforms its competitors for hard setups of skyline processing, which are also the cases where it is most meaningful to exploit multicore processing.

Summarizing, the main contributions of this paper include:

- A new algorithm for multicore skyline computation based on angle-based partitioning that complies with the partition-execute-merge framework.
- Different partitioning methods that result in improved performance depending on the nature of the underlying datasets.
- Novel techniques for parallelization of the partitioning task, in order to reduce the cost of applying sophisticated partitioning methods.
- An extensive evaluation that compares APSkyline with the state-of-the-art algorithms for multicore skyline processing, which demonstrates that our solution outperforms its competitors for hard setups of skyline processing.

The rest of this paper is organized as follows: Sect. 2 provides an overview of related work. In Sect. 3 we describe APSkyline and the partition-execute-merge framework, while in Sect. 4 we describe in detail how partitioning is performed in APSkyline. Our experimental results are presented in Sect. 5. Finally, in Sect. 6 we conclude the paper.

2 Related Work

Since the first paper on skyline processing in databases appeared [4], there have been a large number of papers on the topic, and for a brief overview of current state-of-the-art we refer to [6]. As shown in previous papers on main-memory skyline computing with no indexes available [9, 11, 12], the best-performing algorithm is SSkyline (also known

as Best) [14], and this is also used as a building block in most approaches to multicore skyline processing.

For efficient multicore computing of skylines, there are two state-of-the-art algorithms, PSkyline [9, 11] and ParallelBNL [12]. As shown in [12], PSkyline usually performs better than ParallelBNL in the case of unlimited memory available, but in a memory-constrained context, ParallelBNL performs better.

The PSkyline algorithm [9, 11], is a divide-and-conquer-based algorithm optimized for multicore processors. In contrast to most existing divide-and-conquer (D&C) algorithms for skyline computation that divide into partitions based on geometric properties, PSkyline simply divides the dataset linearly into N equal sized partitions. The local skyline is then computed for each partition in parallel using SSkyline, and the local skyline results subsequently merged. In the evaluation by Im et al. [9] of PSkyline and parallel versions of the existing algorithms BBS and SFS, PSkyline consistently had the best utilization of multiple cores.

In [12], Selke et al. suggest a parallel version of BNL using a shared linked list for the skyline window. This is a straightforward approach, where a sequential algorithm is parallelized without major modifications. However, there are some issues related to concurrent modification of the shared list. Three variants of list synchronization are suggested in the article: continuous, lazy, and lock-free. With continuous locking, each thread will acquire a lock on the next node before processing, lazy locking [7] only locks nodes that should be modified (or deleted), while lock-free is an optimistic approach that does not use any locking. Lazy and lock-free variants need to verify that iterations have been correctly performed and restart iterations that fail. In their evaluation, the lazy locking scheme is shown to be most efficient, and in the evaluation in Sect. 5, we refer to parallel BNL using the lazy locking scheme.

Relevant for our work on multicore skyline computation is also parallel and distributed skyline processing [1, 8, 15]. However, these techniques are based on the assumption that communication between processing units (servers) is very costly compared to local processing, thus making them unsuitable for direct adaption in a multicore context. There has also been other recent work on computing skylines using specialized parallel hardware, e.g., GPU [3] and FPGA [16].

3 The Partition-Execute-Merge Framework

Parallel query processing typically follows the *partition-execute-merge* framework, where three distinct phases are used to split the work to multiple workers (cores or servers, depending on the parallel setting):

1. **Partitioning phase:** The input data is usually split (partitioned) in N non-overlapping subsets S_i following the concept of horizontal partitioning, and each subset is assigned to a worker for processing.
2. **Execute phase:** Corresponds to the actual processing of each individual partition and entails a query processing algorithm that operates on the input data of the partition and produces candidate results for inclusion in the final result set.
3. **Merging phase:** Depending on the type of query, the merging phase may discard some candidate results from the final result by comparing with candidates from

Algorithm 1. Partition-execute-merge

Require: P is the input relation, N is number of partitions
Ensure: SKY_P is the skyline of P
$S_1, \ldots, S_N \leftarrow$ *Partition*(P, N)
$L_1, \ldots, L_N \leftarrow$ *SSkyline*$(S_i, \ldots, S_N)$
$SKY_P \leftarrow$ *SkylineMerge*$(L_1, \ldots, L_N)$

other partitions. For example, in the case of a range query, all candidate results from each partition will be part of the final result set, thus the work of the merging phase is minimal. In contrast, in the case of the demanding skyline query, points from different partitions may dominate other points, which must be removed from the skyline set. Thus, the merging phase of skyline queries entails significant processing cost.

In the following, we describe in more detail each phase of the framework in the context of skyline query processing using APSkyline, and elaborate on the individual objectives of each phase. Algorithm 1 describes our overall approach.

3.1 Partitioning Phase

The objective of the partitioning phase is to divide input data in such a way that (a) individual partitions produce local skyline sets of low cardinality, and (b) the number of points in each partition is balanced. The former goal makes intra-partition query execution particularly efficient and also reduces the cost of the merging phase, while the latter avoids having an individual worker that is assigned a larger portion of work that may lead to delayed execution and initiation of the merging phase. We will describe this in more detail in Sect. 4.

3.2 Execute Phase

In this phase, an efficient skyline processing algorithm is employed to produce the local skyline set of a given data partition. In principle, index-based skyline algorithms are not applicable due to the additional overhead that would be required by index construction. As a consequence, candidate algorithms are selected from the non-indexed family of skyline algorithms. Obviously, the objective of this phase is to rapidly produce the local skyline set of the partition.

In our approach, each thread executes an instance of SSkyline [14] using its private partition as input to compute the local skyline independently. In short, SSkyline takes an input a dataset P containing $|P|$ tuples as input, and returns the skyline set of P. The skyline is computed using two nested loops and three indices: $head$, $tail$, and i. Intuitively the inner loop searches for the next skyline tuple, while the outer loop repeats the inner loop until all skyline tuples have been found. Fig. 1 shows an example run of SSkyline. In the first iteration $head$ points to the first tuple, i to the second, and $tail$ to the last. Confirmed skyline tuples are placed left of $head$ and colored black, confirmed non-skyline tuples are placed to the right of $tail$ and marked with gray, while tuples

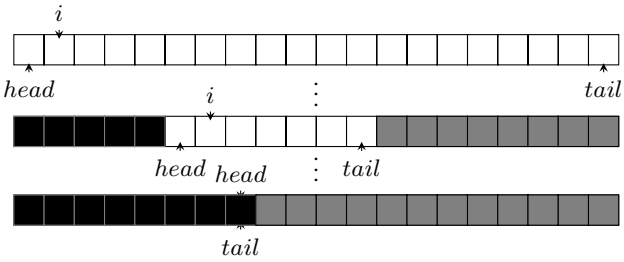


Fig. 1. SSkyline example. Black boxes are part of the skyline, white boxes are undetermined, and gray boxes are dominated (i.e., gray boxes are not part of the skyline).

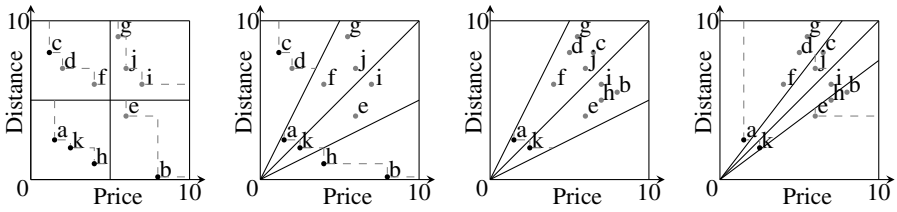
in-between are still under consideration and will at some point be pointed to by $head$ if they are part of the skyline. Each iteration of the outer loop confirms one skyline tuple by moving $head$ to the right, and the inner loop may discard many non-skyline tuples by moving $tail$ to the left. When $head = tail$, the algorithm terminates and returns all skyline tuples. At this point, head and tail point to the last skyline tuple, while i has been discarded.

3.3 Merging Phase

In the merging phase, the local skyline sets produced by the workers need to be combined in order to identify dominated points and exclude them from the final result set. In our approach, we perform parallel skyline merging using the *pmerge* algorithm from [11]. In short, the local skyline sets are merged to the final result one by one, starting with merging two of the local skyline set into a temporary skyline set, and then merging the other local skyline sets into this set one by one. The merging of the temporary skyline set and local skyline sets is performed in parallel, allowing threads to fetch tuples from the local skyline set in a round robin fashion until all local skyline sets have been merged.

4 Partitioning in APSkyline

In contrast to previous approaches that perform random partitioning naively, we intend to exploit geometric properties of the data during the partitioning. The objective of our approach is twofold: (1) reducing the time of the execute phase by assigning equally the points to threads and in a way that is beneficial for skyline processing, and (2) minimize the cost of the merge phase by reducing as much as possible the number of local skyline points. Improving the efficiency of the execute and merge phase, may entail additional processing cost for the partitioning phase, which will not dominate the overall processing cost for large and demanding datasets. In this section, we describe in more detail how to perform efficient angle-based partitioning for multicore skyline computation, using the partitioning techniques presented in [15] as a starting point.



(a) Grid partitioning

(b) Angle partitioning, equi-volume

(c) Angle partitioning, skewed equi-volume

(d) Angle partitioning, dynamic

Fig. 2. Partitioning example using 4 partitions. To the left, (a) and (b) illustrate benefits of angle-based compared to grid-based partitioning (skyline points c, a, k, h, b). To the right, (c) and (d) illustrate benefits of dynamic angle-based partitioning (skyline points a and k).

In the following, in Sect. 4.1, we first describe basic partitioning schemes for skyline processing, and then present our two variants of angle-based partitioning (*sample-dynamic* and *geometric-random*) that are appropriate for multicore systems and we employ in this work. Then, in Sect. 4.2, we show how to utilize parallel compute power not only in phases that compute the actual skyline, but also in the partitioning phase.

4.1 Partitioning Methods

Basic Techniques. Several partitioning techniques exist for parallel skyline computation. The most straightforward is simply dividing the dataset in N partitions at random without considering geometric properties of the data. In fact, this is the approach followed by the current state-of-the-art in multicore skyline processing [9, 11, 12]. However, this often results to large-sized local skyline sets with many points that do not belong to the final skyline set and leads to high processing cost for the execute and merge phase. In more details, if the points are randomly distributed, each partition follows the initial data distribution of the dataset, which means that only for some data distribution the skyline algorithm will perform efficiently [15].

A straightforward approach taking geometric properties into account is grid-based partitioning, however as illustrated in Fig. 2(a) this will result in data partitions that do not contribute to the overall skyline set, resulting in a lot of redundant processing. A more efficient approach is angle-based partitioning, first proposed by Vlachou et al. [15]. As can be seen in Fig. 2(b), (equi-volume) angle-based partitioning produces partitions whose local skyline sets are more likely to be part of the overall skyline set. For details on calculation of partition boundaries (equi-volume) based on the volume of the partitions we refer to [15].

Sample-Dynamic Partitioning. Even though equi-volume angle-based partitioning is applicable for multicore systems, as the boundaries are defined by equations and not by the data itself, the quality of the partitioning highly depends on the data distribution.

One possible problem of the equi-volume angle-based partitioning approach is the uneven sizes of produced partitions in case of skewed datasets, as illustrated in Fig. 2(c) where some partitions are empty. For this purpose, *dynamic* angle-based partitioning can be employed. Nevertheless, dynamic angle-based partitioning in multicore systems entails high processing cost and cannot be applied in a straightforward manner.

In dynamic partitioning, a maximum number of points for each partition is defined, as illustrated in Fig. 2(d). Initially, during partitioning, there is only one partition. During partitioning, when the maximum limit is reached for a partition, the actual partition is split into two. However, in a dynamic partitioning scheme, each partition split induces an expensive redistribution cost. More specifically, each time a partition is split $\frac{n_{max}}{2}$ (where n_{max} is the maximum partition size) or more tuples need to be moved from one memory location to another. This is reasonable in a parallel or distributed environment where IO-operations (including communication between nodes) are the dominating factor. However, in a shared-memory system, such a method requires a significant amount of the overall runtime. Therefore, we adopt a sample-based technique where a small sample of the data is used in order to determine the partitioning boundaries, before actual partitioning is performed.

In the *sample-dynamic* partitioning scheme, a configurable percentage s of the dataset is used to pre-compute the partitioning boundaries. To increase the likelihood of picking representative sample points, we propose to choose samples uniformly at random.

Geometric-Random Partitioning. In some cases, neither equi-volume nor sample-dynamic partitioning are able to achieve a fair workload, thus APSkyline will not be able to use the available parallel computing resources optimally. For example, input with equal rows may cause a skewed workload, and cannot be fairly distributed by geometric partitioning alone. To handle such cases, we propose a hybrid partitioning technique that utilizes geometric properties in combination with random partitioning in order to prioritize a fair workload.

The idea is to modify geometric partitioning schemes like equi-volume and sample-dynamic by having a maximum size of a partition, for example $|P|/N$. During partitioning, points mapped to a partition that is already full are allocated to another, randomly selected, partition. Note that in the case of geometric-random combined with sample-dynamic partitioning, the geometric-random modification is only applied during the final partitioning, not during processing of the sample for calculation of partition bounds.

4.2 Parallelism in the Partitioning Phase

In contrast to algorithms directed at parallel and distributed systems, a shared-memory algorithm needs to have a highly optimized partitioning phase. Thus, in the following we present techniques for parallelizing our partitioning methods. Algorithm 2 shows the parallel partitioning algorithm for the case of equi-volume partitioning. We use *N* threads in order to partition a relation into *N* partitions. Obviously, we need some way of determining which points each thread should distribute. This can be done in a round robin fashion, or using a linear partitioning strategy to define read boundaries without physically partitioning points. We use the linear strategy. To split data

Algorithm 2. ParallelEquiVolumePartitioning

Require: P is the input relation, N is number of partitions/threads
Ensure: $S_i, 0 \leq i < N$ contains the final partitioning
  partitionBounds $\leftarrow$ *determinePartitionBounds*()
  parallel for $t = 0$ **to** $N - 1$ **do** ▷ Distribute work over N threads
    for $k = t\frac{|P|}{N}$ **to** $(t+1)\frac{|P|}{N}$ **do**
      $p \leftarrow P[k]$
      $i \leftarrow$ *MapPointToPartition*$(p,$ *partitionBounds*$)$
      $lock(S_i)$
      $S_i \leftarrow S_i \cup \{p\}$
      $unlock(S_i)$

Algorithm 3. ParallelDynamicPartitioning

Require: P is the input relation, N is number of partitions/threads s is sample fraction
Ensure: $S_i, 0 \leq i < N$ contains the final partitioning
  nextPartID $\leftarrow 0$
  partitionBounds $\leftarrow$ *determineInititialBounds*()
  for $t = 0$ **to** $s|P|$ **do** ▷ Sample size
    $p \leftarrow P[$*rand*$(0, |P| - 1|)]$ ▷ Random point
    $i \leftarrow$ *MapPointToPartition*$(p,$ *partitionBounds*$)$
    $S_i \leftarrow S_i \cup \{p\}$
    if S_i is full **then**
      nextPartID $\leftarrow$ *nextPartID* $+ 1$
      split S_i into S_i, S_{next}
      partitionBounds $\leftarrow$ *determineNewBounds*()
  parallel for $t = 0$ **to** $N - 1$ **do** ▷ Distribute work over N threads
    for $k = t\frac{|P|}{N}$ **to** $(t+1)\frac{|P|}{N}$ **do**
      $p \leftarrow P[k]$
      $i \leftarrow$ *MapPointToPartition*$(p,$ *partitionBounds*$)$
      $lock(S_i)$
      $S_i \leftarrow S_i \cup \{p\}$
      $unlock(S_i)$

into 2 partitions, thread 1 processes tuples $[1, \ldots, |P|/2]$ and thread 2 processes tuples $[|P|/2 + 1, \ldots |P|]$. This ensures no need for locking during read. All threads place tuples into multiple shared collections (partitions) so here locks are needed. However, we emphasize that in contrast to locks used for concurrency control in, e.g., database systems, this is low-overhead locks, and typically implemented as spinlocks. In practice, some collisions will occur and threads will sometimes have to wait for locks. However, because the time needed to write result tuples is much smaller than the time required to perform the calculations for mapping a point to partition, lock waiting time is not significant.

In contrast to the equi-volume scheme where the boundaries are pre-determined, for the sample-dynamic partitioning scheme the boundaries must be computed first. In this case we suggest a two-step process, where partitioning boundaries are calculated sequentially using a certain percentage of the input relation (i.e., a sample) before parallel

partitioning is performed, as described in Algorithm 3. In the first step, the bounds are computed and then in the second step the data points are partitioned. Finally, it is straightforward to modify Algorithm 3 for supporting the geometric-random partitioning scheme.

5 Experimental Evaluation

In this section, we present the results of the experimental evaluation. All our experiments are carried out on a machine with two Intel Xeon X5650 2.67GHz six-core processors, thus providing a total of 12 physical cores at each node. Each processor can run up to 12 hardware threads using hyper-threading technology. Each core is equipped with private L1 and L2 caches, and all cores on one die share the bigger L3 cache. All algorithms are implemented in Java.

5.1 Experimental Setup

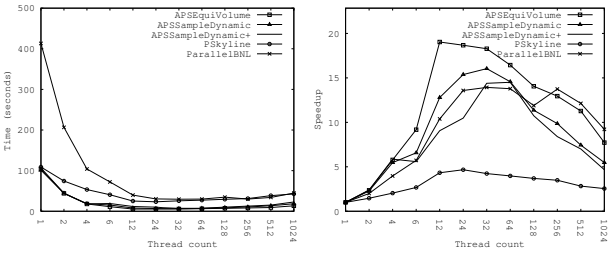
Datasets. For the dataset P we employ both real and synthetic data collections. The synthetic sets are: (1) uniform (UN), (2) correlated (CO), and (3) anti-correlated (AC). For UN dataset, the values for all d dimensions are generated independently using a uniform distribution. The CO and AC datasets are generated as described in [4]. The real datasets are: (1) NBA which is a 5-dimensional dataset containing approximately 17K entries, where each entry records performance statistics for a NBA player, and (2) ZILLOW5D, a 5-dimensional dataset containing more than 2M entries about real estate in the United States based on crawled data (from www.zillow.com), and where each entry includes number of bedrooms and bathrooms, living area, lot area, and year built.

Algorithms. We compare our new algorithms against the current state-of-the-art multicore algorithms PSkyline [9, 11] and ParallelBNL [12] (implementations are based on source code made available by Selke et al. [12]). We implemented three variants of our algorithm (APSkyline) that differ based on the variant of angle-based partitioning employed:

- APSEquiVolume: Equi-volume angle-based partitioning.
- APSSampleDynamic: Sample-dynamic partitioning with sample size $s = 1\%$.
- APSSampleDynamic+: Sample-dynamic partitioning in combination with the geometric-random modification, with limit for each partition $\frac{|P|}{N}$, sample size $s = 1\%$.

Measurements. Our main metric is the runtime of each algorithm. Each test is executed 10 times and median values are used when reporting results. We perform one dry-run before taking any measurements in every experiment. Additionally, we measure variance, minimum, and maximum values in order to ensure that tests are sufficiently accurate. For synthetic datasets, new input is generated for each of the 10 executions, in order to factor out the effect of randomization.

Parameters. We vary the following parameters: dimensionality (d=2-5), cardinality ($|P|$=50K-15M), number of threads (partitions) (N=1-1024), and data distribution



(a) Runtime (b) Speedup

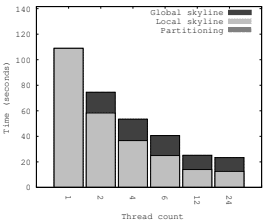
Fig. 3. Comparison of algorithms running with a different number of threads

(AC,CO,UN,NBA,ZILLOW5D). Default parameters are size of 5M 5-dimensional tuples of AC distribution. As also observed in [15], the anti-correlated dataset is most interesting, since the skyline operator aims to balance contradicting criteria [10, 15]. Moreover, anti-correlated distributions are closer to many real-life datasets according to [13]. Combined with the fact that multicore processing is typically employed for expensive setups of query processing, AC distribution is the setup that makes sense in the multicore context.

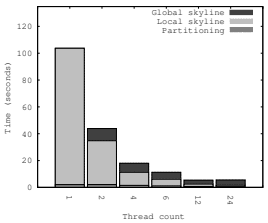
5.2 Experimental Results

Effect of Thread Count. Fig. 3(a) shows the results in the case of an AC dataset as we increase the number of threads from 1 to 1024. We expect to reach peak performance at 24 threads, which is the maximum number of hardware threads available (12 physical cores + hyper-threading). As number of threads increase beyond 24, we expect that performance will gradually decrease due to increased synchronization costs without additional parallel compute power. A main observation is that the APSkyline variants consistently outperform the competitor algorithms, which is a strong witness for the merits of our approach. For a low thread count ParallelBNL is inefficient compared to all the other D&C based algorithms. This is most likely due to the fact that D&C based algorithms use SSkyline for local skyline computation, which uses an array for storing results. The linked list used in ParallelBNL is not as memory-efficient as an array structure for sequential computations. Unsurprisingly, there is no significant performance difference between variations of APSkyline (Fig. 3(a)).

In Fig. 3(b), we measure the speedup of each algorithm. The speedup for each algorithm is relative to the same algorithm run with one thread, not to a common reference point, in order for the results to be easily compared to related work [11, 12]. We observe that APSkyline achieves super-linear speedup for up to 12 threads. Obviously, super-linear speedup cannot be explained by parallelism alone, we therefore attribute positive results to a combination of an increased parallel compute power, an increase in high-level cache (each core contributes with its private cache), and smaller input cardinalities for SSkyline. ParallelBNL has a good speedup as thread count is increased.

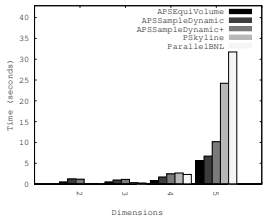


(a) PSkyline



(b) APSEquiVolume

Fig. 4. Segmented runtime for PSkyline and APSEquiVolume



(a) Dimensionality

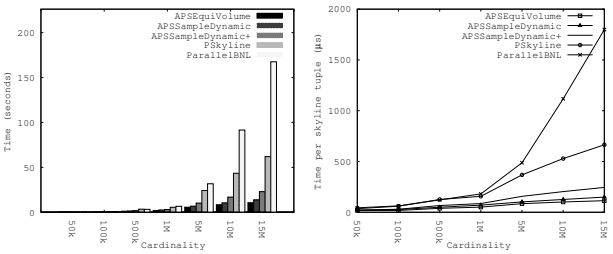
Fig. 5. Comparison with varying dimensionality

This is in line with results presented in [12], and shows that a basic algorithm can be quite effective when parallel compute power and low-cost synchronization constructs are available. PSkyline shows a modest speedup compared to the other algorithms.

Fig. 4 shows the segmented runtime for PSkyline and APSEquiVolume (the two best-performing algorithms). Local skyline computation shows diminishing performance gains as the available parallel compute power increases. Partitioning refers to the time needed for the partitioning phase (Sec. 3.1), local skyline for the time of the execute phase (Sec. 3.2) and global skyline to the time of the merging phase (Sec. 3.3). Local skyline processing gets more efficient as the number of threads increase for both approaches, but in the case of APSEquiVolume the processing cost reduces more rapidly. This is due to the employed geometric partitioning, that alters the data distribution of the points assigned to each thread in a way that the required domination tests are fewer. In contrast, a random partitioning technique is expected to assign points that follow the anti-correlated data distribution to each thread, which leads to a more demanding local processing. Due to the geometric partitioning, APSEquiVolume results also to fewer local skyline tuples, which in turn leads to a smaller processing cost of the merging phase compared to PSkyline.

In summary, APSkyline clearly outperforms all other algorithms. When all cores are in use (24 threads), APSkyline is *4.2 times faster than PSkyline* and *5.2 times faster than ParallelBNL*. Fig. 4 also shows that the partitioning technique used in APSkyline is more expensive than the one used in PSkyline. Nevertheless, the time spent for partitioning is negligible compared to benefits attained in the subsequent phases. APSkyline is able to utilize parallel compute power in every phase as shown by Fig. 4.

Effect of Data Dimensionality. Fig. 5 shows the obtained results when increasing the number of dimensions from 2 to 5. For a dimensionality of 3 or less, ParallelBNL is the most efficient algorithm, while APSSampleDynamic+ is the least efficient. For a dimensionality of 4, APSEquiVolume and APSSampleDynamic outperform other algorithms by a small margin. Finally, all variations of APSkyline significantly outperform ParallelBNL and PSkyline for 5-dimensional datasets. In contrast to earlier algorithms, APSkyline scales well with dimensionality. It should be emphasized that size of the



(a) Runtime (b) Runtime per skyline tuple

Fig. 6. Comparison of algorithms running for varying data cardinality

skyline set increases rapidly with the dimensionality of the dataset, making skyline processing for higher dimensional data more demanding. This experiment verifies that for hard setups, as in the case of high dimensionality, our algorithms outperform the competitors and (more importantly) the benefit increases for higher values of dimensionality.

Effect of Data Cardinality. In Fig. 6(a), we examine how the algorithms scale for increased size of dataset. We observe that APSkyline achieves the best runtime for all input sizes. In particular, we notice that APSEquiVolume is *15.8 times faster than ParallelBNL* and *5.9 times faster than PSkyline* with 15M input tuples. Fig. 6(a) clearly depicts that APSkyline variants are robust when increasing the data cardinality. Moreover, this experiment shows that even for small-sized AC datasets (which contain a large percentage of skyline tuples), the setup is challenging thus both ParallelBNL and PSkyline demonstrate sub-optimal performance.

Fig. 6(b) shows the time used per skyline tuple by each algorithm. It is evident that ParallelBNL and PSkyline do not handle high dataset cardinalities well. Time used per skyline tuple should ideally be unchanged as cardinality increase. However, the merging phase requires pairwise comparisons between local skyline tuples, which in turn lead to a quadratic and not linear behavior of the skyline algorithms. As the number of local skyline tuples increase with the dataset cardinality, it is expected that the time per skyline tuple increases due to the pairwise comparisons. In this regard, APSkyline is quite successful. Processing time per skyline tuple increases very slowly compared to ParallelBNL and PSkyline, and this is an excellent example of the ability of an angle-based partitioning scheme to eliminate non-skyline tuples early.

Effect of Data Distribution. Fig. 7 shows the results for different synthetic data distributions and depicts the segmented runtime for each algorithm. First, we observe that skyline processing over the AC dataset is much more demanding for any algorithm (5000-30000 msec) than skyline processing over the UN or CO dataset (100-1300 msec). This verifies our claim that the AC dataset is a more typical use-case of multicore processing, which primarily targets the case of expensive query operators. AP-

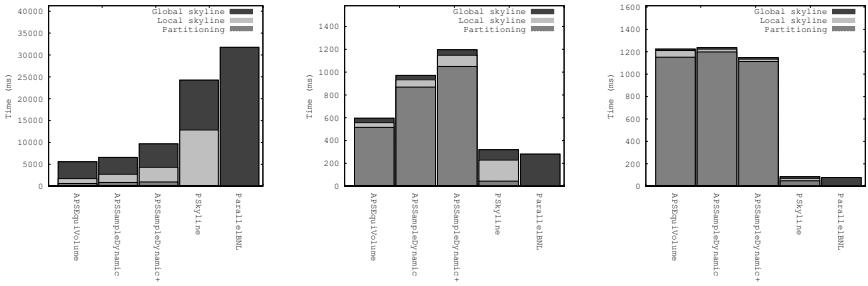


Fig. 7. Segmented runtime for synthetic datasets (anti-correlated to the left, independent in middle, and correlated to the right)

Skyline is significantly faster than its competitors for the challenging AC dataset. The most efficient variant (APSEquiVolume) is almost 5 times better than PSkyline and 6 times better than ParallelBNL. Despite the fact that APSkyline is significantly faster for hard setups (e.g., AC) where the number of skyline points is high, in the case of easy setups (e.g., UN or CO) the cost of partitioning of APSkyline dominates its runtime thus rendering the competitor algorithms more efficient.

When comparing the variants of APSkyline, we observe that equi-volume partitioning is most efficient. Due to the synthetic data generation, the dataset is fairly distributed by the equi-volume scheme for the AC and UN datasets. In addition, the partitioning is clearly more efficient for APSEquiVolume, since no sampling is used, and the partition boundaries are simply determined by equations independently of the underlying data. Thus, the small gain in the performance of computing the local and (global) overall skyline sets is dominated by the additional cost of the partitioning, rendering APSEquiVolume the best variant for synthetic datasets.

Real Datasets. Fig. 8 shows the obtained results for the real-life datasets. First, Fig. 8(a) depicts the statistics for the NBA dataset. Recall that this dataset is a small dataset containing approximately only 17K tuples. Combined with the fact that NBA is fairly correlated [5] means that NBA is not a very challenging case for skyline computation. Thus, Fig. 8(a) clearly depicts that the overhead of partitioning is too high compared to the total processing cost. PSkyline achieves the best performance for NBA, even though D&C based algorithms show similar performance for the local skyline computation and the merging phase. However, APSkyline spends much time partitioning and is therefore less efficient than PSkyline. Moreover, the equi-volume partitioning scheme is outperformed by all other partitioning schemes. This is attributed to the fact that the dynamic strategy is better tailored for real-life datasets which are not symmetric as the synthetic datasets. In case of symmetries, an equi-volume partitioning can distribute the work fairly. In contrast, in a real-life dataset, a (sample-)dynamic partitioning scheme is more robust than a fixed scheme that does not adapt to its input.

For the ZILLOW5D dataset, the best-performing algorithm is APSSampleDynamic+, thus demonstrating the usefulness of the geometric-random modification. ParallelBNL is outperformed in an order of magnitude by all other algorithms. In fact, APSSample-

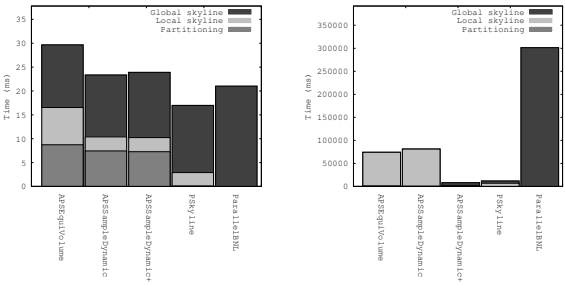


Fig. 8. Segmented runtime for real-life datasets (NBA to the left, ZILLOW5D to the right)

Dynamic+ is 36 times faster than ParallelBNL in this case. Additionally, we observe that PSkyline and APSSampleDynamic+ spend significantly less time in the local skyline computation phase than APSEquiVolume and APSSampleDynamic. The reason for APSEquiVolume and APSSampleDynamic spending so much time in the local skyline computation phase when processing the ZILLOW5D dataset is lack of a fair work distribution. We observed that most points were placed in only a few partitions, causing the majority of threads being idle. APSSampleDynamic and PSkyline do not have this problem, as they always divide data fairly. Nevertheless, APSSampleDynamic+ is able to outperform PSkyline with a factor of 1.4 using an angle-based partition technique in combination with random partitioning.

Discussion. In summary, our novel algorithm APSkyline based on angle-partitioning outperforms existing approaches for the most time-consuming datasets, while ParallelBNL and PSkyline excelled for simpler cases. By taking into account that the anti-correlated dataset is most interesting for skyline queries, since the skyline operator aims to balance contradicting criteria combined with the fact that multicore processing is typically employed for expensive setups of query processing, it highlights the value of our approach. Moreover, PSkyline performed slightly better than APSkyline variants for a small real-life dataset and was quite efficient for a large real-life dataset. Nevertheless APSSampleDynamic+ was able to reduce runtime by approximately 30% compared to PSkyline for the large real-life dataset. Thus, the APSkyline variants outperformed the existing approaches for all demanding datasets.

6 Conclusions

In this paper, we have presented APSkyline, a new approach for multicore skyline computing. The use of angle-based partitioning increases the degree of pruning that can be achieved in the execute phase, thus significantly reducing the number of candidate points that need to be checked in the final merging phase. As shown by our experimental evaluation, APSkyline is extremely efficient for hard cases of skyline processing, where we significantly outperform the previous state-of-the-art approaches.

References

1. Afrati, F.N., Koutris, P., Suciu, D., Ullman, J.D.: Parallel skyline queries. In: Proc. of ICDT (2012)
2. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core CPUs. In: Proc. of SIGMOD (2011)
3. Bøgh, K.S., Assent, I., Magnani, M.: Efficient GPU-based skyline computation. In: Proc. of DaMoN (2013)
4. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proc. of ICDE (2001)
5. Chan, C.-Y., Jagadish, H.V., Tan, K.-L., Tung, A.K.H., Zhang, Z.: On high dimensional skylines. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 478–495. Springer, Heidelberg (2006)
6. Chomicki, J., Ciaccia, P., Meneghetti, N.: Skyline queries, front and back. SIGMOD Record 42(3), 6–18 (2013)
7. Heller, S., Herlihy, M.P., Luchangco, V., Moir, M., Scherer III, W.N., Shavit, N.N.: A Lazy Concurrent List-Based Set Algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006)
8. Hose, K., Vlachou, A.: A survey of skyline processing in highly distributed environments. VLDB J. 21(3), 359–384 (2012)
9. Im, H., Park, J., Park, S.: Parallel skyline computation on multicore architectures. Inf. Syst. 36(4), 808–823 (2011)
10. Morse, M., Patel, J.M., Jagadish, H.: Efficient skyline computation over low-cardinality domains. In: Proc. of VLDB (2007)
11. Park, S., Kim, T., Park, J., Kim, J., Im, H.: Parallel skyline computation on multicore architectures. In: Proc. of ICDE (2009)
12. Selke, J., Lofi, C., Balke, W.T.: Highly scalable multiprocessing algorithms for preference-based database retrieval. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA 2010. LNCS, vol. 5982, pp. 246–260. Springer, Heidelberg (2010)
13. Shang, H., Kitsuregawa, M.: Skyline operator on anti-correlated distributions. PVLDB 6(9), 649–660 (2013)
14. Torlone, R., Ciaccia, P.: Finding the best when it's a matter of preference. In: Proc. of SEBD (2002)
15. Vlachou, A., Doulkeridis, C., Kotidis, Y.: Angle-based space partitioning for efficient parallel skyline computation. In: Proc. of SIGMOD (2008)
16. Woods, L., Alonso, G., Teubner, J.: Parallel computation of skyline queries. In: Proc. of FCCM (2013)