Efficient Parallel Skyline Processing using Hyperplane Projections

Henning Köhler[1]
henning@itee.uq.edu.au

Jing Yang[2]
jingyang@ruc.edu.cn

Xiaofang Zhou[1,2]
zxf@uq.edu.au

[1] School of Information Technology and Electrical Engineering, The University of Queensland, Australia
[2] School of Information, Renmin University of China
Key Labs of Data Engineering and Knowledge Engineering, Ministry of Education, China

ABSTRACT

The skyline of a set of multi-dimensional points (tuples) consists of those points for which no clearly better point exists in the given set, using component-wise comparison on domains of interest. Skyline queries, i.e., queries that involve computation of a skyline, can be computationally expensive, so it is natural to consider parallelized approaches which make good use of multiple processors. We approach this problem by using hyperplane projections to obtain useful partitions of the data set for parallel processing. These partitions not only ensure small local skyline sets, but enable efficient merging of results as well. Our experiments show that our method consistently outperforms similar approaches for parallel skyline computation, regardless of data distribution, and provides insights on the impacts of different optimization strategies.

1. INTRODUCTION

There are many applications where a user is interested in viewing the 'best' objects chosen from a large collection, based on multiple criteria, e.g. price and mileage for used cars. There are multiple ways to approach this problem. Top-k queries [5] require a user to define a ranking function over the object collection, and return the k top-ranked objects. In contrast, skyline queries, first introduced by [4], only require users to express their preferences for each domain of interest, e.g. price and mileage should both be low. They then return all objects for which no clearly better object exists, i.e., an object which is at least as good on every domain and strictly better in at least one. Figure 1 shows such a skyline (containing four points) for used cars. Shaded areas indicate values for which strictly better points exists.

The resulting skyline can provide a user with a better understanding of the trade-offs involved, without requiring any precise ranking functions to be specified. This flexibility makes skylines preferable to top-k approaches for many scenarios, and has caused them to receive much attention in recent years.

SIGMOD'11, June 12–16, 2011, Athens, Greece.

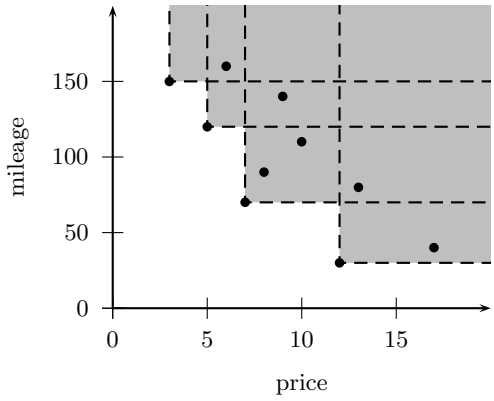


Figure 1: Skyline of used cars

As data sets used for skyline processing are often huge, computation can be expensive, and efficient algorithms are vital for applications requiring fast response times. With the advance of multi-core architectures and other parallel computing platforms, parallel skyline algorithms offer a new way to boost performance. In this paper we will develop such a parallel algorithm, with focus on the initial data partitioning. The key idea here is to group points based on their direction from the origin, thus increasing the likelihood of dominance between them. This is achieved by projecting points onto a hyperplane, and ensures that local skyline sets are small. At the same time, merging of result sets can be done efficiently.

Our approach is closest to the angle-based partitioning approach of [23], which also generates small local skylines, but is consistently faster (by up to one order of magnitude), especially for anti-correlated data sets. The improvement hails from a more sophisticated merge step, as well as faster initial partitioning. In addition, we investigate the impact of an approximate skyline pre-processing step, which is generally applicable, and has already proven helpful for sequential skyline computation [12].

The rest of this paper is organized as follows. In Section 2 we give a brief overview of existing work on distributed and parallel skyline computation. The most closely related approach of [23] is discussed in more detail in Section 3, where we develop our plane-project-parallel-skyline (PPPS) algorithm. Experimental evaluation of our and related approaches can be found in Section 4. Section 5 concludes.

2. RELATED WORK

The skyline operation is useful for extracting interesting information from multidimensional databases. It originates from the maximal vector problem in computational geometry [15]. Those early algorithms, including [2, 16], are in-memory algorithms which are only suitable for small datasets. In the original database-oriented paper by Borzsonyi et al. [4] where the concept of the skyline operator is introduced for large databases, a number of efficient external memory algorithms are also proposed, including the block-nested-loop (BNL) algorithm that scans the dataset while employing a bounded buffer for tracking the points that cannot be dominated by other points in the buffer, and the divide-and-conquer (DC) algorithm that recursively partitions the dataset until each partition is small enough to fit in memory to compute the local skyline for each partition followed by a merge step to form the global skyline. There exist numerous efficient skyline query processing approaches, such as the bitmap-based progressive algorithm [22], the sorting-and-filtering approach [6, 12], the nearest neighbor based approach [14], and the I/O optimal R-tree based branch-and-bound algorithm [17, 18].

Due to its high processing cost, there has been a growing interest lately in distributed and parallel skyline query processing. Deviating from centralized skyline query processing approaches, a new approach for computing skyline objects over distributed sources is first presented by Balke et al. [1]. It supports skyline operations over web databases where data is vertically partitioned and each site provides one attribute of the data object. Skyline points are calculated per site and reported to the user at a central point. The algorithm first retrieves values in every dimension from remote data sites using sorted access in round-robin fashion on all dimensions until all dimension values of an object, called the terminating object, have been retrieved. Then those non-skyline objects will be filtered from all those objects with at least one dimension value retrieved. The problem setting for this work is specialized, and the use of a central point can limit the scale of distribution.

There are several data partitioning based approaches for skyline computation in peer-to-peer environment. Wang et al. [24] developed the Skyline Space Partitioning (SSP) approach to compute skylines on a tree-structured P2P platform. That approach uses the z-curve method to partition the multidimensional data space into linearly ordered regions which can be mapped to different peer nodes according to the underlying P2P protocols. As in most partition-based parallel processing schemes, load-balancing is a critical issue. Under the SSP approach, a small number of peers (those that are allocated with partitions close to the origin of the axes) can often take much heavier workload than other peers, while many peers do not contribute to the final skyline results.

Wu et al. [25] are among the first to address the problem of parallel skyline query execution over a large number of machines by leveraging on content-based data partitioning. By using the query range to recursively partition the data region on every data site involved, and encoding each involved sub-region dynamically, their method avoids accessing sites not containing potential skyline points and can report correct skyline points progressively. The proposed algorithm named DSL [25] horizontally partitions data across different machines, i.e. each machine stores a subset of the entire data record set. One advantage of this approach is that it provides incremental scalability, where its performance can be improved by adding additional machines to the cluster. Their system can automatically balance the load by distributing objects to the new nodes. This work, however, requires each node to start the skyline computation on its data after receiving the results of other nodes based on the partial order. Therefore, its parallelism is limited to the pipeline fashion. Differing from the previous work, Cui et al. [8] approach parallel skyline computing without the assumption of any overlay availability on top of the original network. They tackle constrained skyline queries in large-scale distributed environments with horizontally partitioned data distribution. The problem of parallel skyline computing is also considered in an environment of one-processor-multiple-disk architecture to enable disk access parallelism and effective pruning using the parallel R-tree [11].

Dehne et al. [10] propose an optimal coarse-grained parallel algorithm for computing skylines in three dimensions; but their approach does not seem to lead to practical algorithms for higher-dimensional point sets. Random data partitioning is used in [7] which uses the divide-and-conquer algorithm by Kung et al. [15] and the branch-and-bound algorithm by Papadias et al. [18] as building blocks. This can ensure a similar data distribution in each partition to the original dataset. Then each machine processes the skyline over its local data using the branch-and-bound algorithm. The main drawback of this random partitioning approach is that the size of the local result sets is not minimized and many points that belong to the local skyline sets do not belong in the final skyline result set.

In contrast to those parallel skyline algorithms designed for a shared-nothing distributed environment where the participating nodes can only communicate only by exchanging messages [13, 21, 25], [19] focuses on exploiting properties specific to multi-core architectures in which participating cores inside a processor share everything and communicate simply by updating the main memory. They proposed two parallel skyline algorithms: a parallel version of the branch-and-bound algorithm [18] and a new parallel algorithm based on skeletal parallel programming, which is already in use by such database programming models as MapReduce [9, 20] and Map-Reduce-Merge [26].

Grid-based data space partitioning has been commonly used in distributed and parallel skyline processing. For example, in [25], the space is partitioned based on CAN [21], whereas in [24] a tree-structured overlay is used to partition the data space. Such grid-based partitioning, however, is not suitable for skyline queries when all partitions are to be examined at different machines in parallel, since many data partitions do not contribute to the final skyline set (you can think that many partitions are actually dominated by other partitions). This can result in significant redundant processing at some nodes as well as higher overall costs for data communication and merger after location skylines are computed. In [23], a novel angle-based space partitioning scheme is proposed to use the hyperspherical coordinates of the data points to alleviate most of the problems found in traditional random and grid partitioning techniques. It first maps the entire dataset from the Cartesian coordinate space into a hyperspherical space, in which the data space is partitioned based on the angular coordinates. While this type of partitioning has some attractive features, it employs

a centralized step to transform the coordinates of each and every point of the dataset to hyperspherical coordinates. We will discuss more details of this approach in the next section when we introduce our ideas.

3. PARALLEL SKYLINE PROCESSING

We will now describe our approach to parallel skyline processing. For parallel algorithms, there is typically a tradeoff. If we spend time to pre-process the data before allocating tasks to different processors, the processing itself and/or the final merge phase may become cheaper. On the other hand, the time used for pre-processing contributes to the overall running time. Finding a good balance which minimizes overall running time is vital.

Computationally, there is a significant difference between cases where the skyline set is small and cases where it is large. For small skyline sets, skyline computation is I/O bound, but it becomes CPU-bound for large skyline sets as the number of dominance-checks between skyline points grows (quadratically in the worst case). To ensure that our algorithm is efficient for both small and large skyline sets, we must combine a number of processing techniques. By focusing on preprocessing steps which can be performed as part of a single scan of the data set for data partitioning, we remain efficient for small skyline sets.

As a first step we attempt to reduce the number of points to consider for further computation. Here we employ fast algorithms for approximating the skyline. That is, we filter out points which we can quickly identify as not being part of the skyline. An approach for approximate skyline computation is discussed in Section 3.1.

Next we must partition our data set. Each partition will then be transferred to one of the available processors for local skyline computation. Here it is vital to partition in such a manner that non-skyline points can be eliminated during local skyline computation as much as possible, i.e., we would like to group points which are likely to dominate each other into the same partition. The angle-based approach for skyline partitioning proposed in [23] achieves this, but requires rather complex and costly conversion of points into hypersphere coordinates, and does not cater for efficient merging. We will show that equally good partitions can be found through hyper-plane projection (which is much simpler and cheaper to compute), and that such projections allow for efficient merge steps. Our partitioning approach is detailed in Section 3.2, while efficient merging of local skylines is discussed in Section 3.3.

Note that the computation of the approximate skyline of P (see Section 3.1) and its partition into $P_1, \ldots, P_n$ can be done together in a single parse of P, thereby minimizing I/O costs for large data sets.

3.1 Approximate Skyline

Computing the exact skyline of a set P can be expensive, as each point may be compared to many other points. However, it is often possible to eliminate the majority of non-skyline points with few comparisons, especially in cases where the skyline set is small. The idea for this is the following: We pick a small set $B \subset P$ of comparison points, then test for every point in P whether it is dominated by a point in B. Those that are dominated can safely be discarded, so that the resulting approximate skyline set is guaranteed to contain all skyline points, but likely some non-skyline points as well. Importantly, for fixed size B (we found $|B| \approx 10$ to work well), computing the approximate skyline can be done in linear time with a single pass over the data set. For details on strategies for selecting B see [12], which employs similar techniques for optimizing sort-based skyline computation. As we want to avoid scanning the data set multiple times, we select B based on a small sample subset of P.

In addition to reducing transport and computation costs, early elimination of non-skyline points has another positive side-effect. As we partition P such that *input* sets are of similar size for each processor, it can easily happen that the respective local skylines differ significantly in size. As the skyline size impacts on computation time, this can cause unbalanced workloads. Removal of non-skyline points tends to strengthen the relationship between input and output (skyline) size, thus reducing the chance for and/or severity of imbalanced loads.

3.2 Partitioning via Hyperplane Projection

For partitioning the data space, we will employ techniques similar to the angle-based partitioning approach from [23]. The central idea in [23] is to transform each point (treated as a vector in some euclidian space) into hypersphere coordinates. Partitioning is then done based on the transformed values, ignoring the distance from the origin. As a result, points which lie in similar direction from the origin, but possibly at different distances, are likely to be assigned to the same partition. Hence points that can dominate a given non-skyline point are likely to lie in the same partition, thus allowing it to be eliminated during local skyline computation. Figure 2 illustrates this approach - on the left the partition in the original space is shown, on the right the same partition after projection and transformation into hypersphere coordinates.

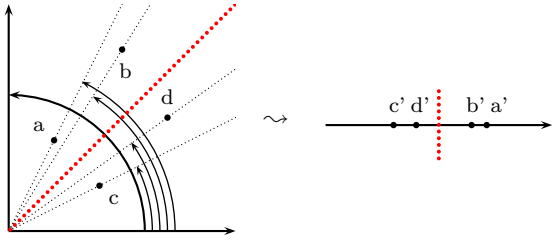


Figure 2: Angle-based Partitioning

Unfortunately, transformation into hypersphere coordinates is time-consuming. This is less noticeable in cases where the skyline is large, but can become a bottleneck if the skyline set is small (and thus computation fast), as these transformations are performed centrally. In our approach, we avoid costly transformation of coordinates into hypersphere coordinates. Instead, we use projection onto the hyperplane $x_1 + \ldots + x_d = 1$. Note that we do not project orthogonally, but by intersecting the line through the origin with the hyperplane. Afterwards, points are allocated to partitions based on their projections. This is illustrated in Figure 3 - the partition of the original space is shown on the left and its hyperplane projection on the right, where it is possible to decide which partition a point belongs to based on a single projected coordinate.

In 2-dimensional space, every angle-based partition is also a hyperplane-based partition and vice versa, although this

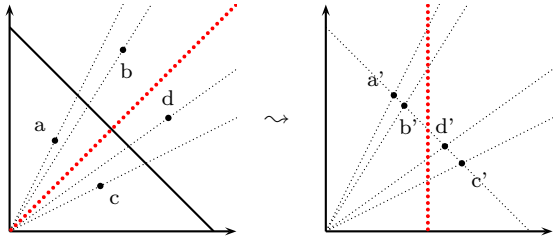


Figure 3: Hyperplane-projection-based Partitioning

fails to hold for dimension higher than 2. Note that unlike the angle-based coordinate transformation, our projection does not affect the dimensionality of the data set. While all projected points lie in a hyperplane, these projections are still represented in the original euclidian space. Importantly, points that are in a dominance relationship are likely to have similar hyperplane projections, and thus are likely allocated to the same partition for local skyline computation. This realizes the driving goals behind the angle-based partitioning approach [23], but unlike transformations into hypersphere coordinates, hyperplane projections are simple and cheap to compute:

$$\begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} \mapsto \begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} \cdot \frac{1}{x_1 + \ldots + x_d}$$

While grid-based partitioning methods, which define a space partition by partitioning each dimension, are not directly applicable to the resulting hyperplane, as the positive section of the hyperplane is not a hypercube, recursive partitioning methods are well suited. Here we split a given dataset into two partitions, based on a single coordinate. Each partition is then split again (using a different coordinate), until the desired number of partitions (=number of processors) is reached. The split conditions (e.g. $x_2 \leq 0.2$) should be chosen such that each partition contains roughly the same number of points for a randomly selected sample set. This can be done efficiently by using one of the well-known selection algorithms such as quick-select, which find the k^{th} smallest element in (worst-case or expected) linear time [3].

Of course, the number of parallel processors available need not be a power of 2 in general. In such a case, whenever we perform a split, we also split the set of processors into two groups of nearly equal size (e.g. 3 and 4 if 7 processors are available), and then select the split point such that the point partitions contain points in the same ratio (e.g. 3:4), and assign each partition to their respective set of processors. A similar approach can be used to deal with processors of different computational power.

Compared to the dynamic space partitioning approach suggested in [23], which repeatedly splits the largest partition into two until the desired number of partitions is reached, our method has two advantages. First, it allows partitioning of the entire data set in a single pass, which reduces I/O costs for large data sets. Second, partitions are very similar in size, whereas the partitions obtained by the method in [23] differ by up to factor two.

EXAMPLE 1. *Consider the set of 3D points P given below:*

$$P = \left\{ \begin{pmatrix} 9 \\ 6 \\ 15 \end{pmatrix}, \begin{pmatrix} 13 \\ 3 \\ 4 \end{pmatrix}, \begin{pmatrix} 15 \\ 5 \\ 5 \end{pmatrix}, \begin{pmatrix} 12 \\ 8 \\ 20 \end{pmatrix} \right\}$$

Projection on the hyperplane $x + y + z = 1$ results in the following projected points:

$$\mathcal{H}(P) = \left\{ \begin{pmatrix} 0.3 \\ 0.2 \\ 0.5 \end{pmatrix}, \begin{pmatrix} 0.65 \\ 0.15 \\ 0.2 \end{pmatrix}, \begin{pmatrix} 0.6 \\ 0.2 \\ 0.2 \end{pmatrix}, \begin{pmatrix} 0.3 \\ 0.2 \\ 0.5 \end{pmatrix} \right\}$$

Imposing a split condition $x_1 \leq 0.4$ partitions P into

$$P_1 = \left\{ \begin{pmatrix} 9 \\ 6 \\ 15 \end{pmatrix}, \begin{pmatrix} 12 \\ 8 \\ 20 \end{pmatrix} \right\} \quad \textit{and} \quad P_2 = \left\{ \begin{pmatrix} 13 \\ 3 \\ 4 \end{pmatrix}, \begin{pmatrix} 15 \\ 5 \\ 5 \end{pmatrix} \right\}$$

Note that rotating dimensions for splitting helps to ensure that the projections of points within a partition are similar in all dimensions, thus increasing the probability that points within the same partition dominate each other.

Finally, the choice of hyperplane $x_1 + \ldots + x_d = 1$ works best if all coordinates have roughly the same range of values. If this is not the case, we can scale dimensions, e.g. by their (estimated) average value $Ave(X_i)$, giving us the transformation

$$\begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} \mapsto \begin{pmatrix} \frac{x_1}{Ave(X_1)} \\ \vdots \\ \frac{x_d}{Ave(X_d)} \end{pmatrix} \cdot \frac{1}{\frac{x_1}{Ave(X_1)} + \ldots + \frac{x_d}{Ave(X_d)}}$$

3.3 Merging Skylines

When merging the partial skyline sets to obtain the final skyline results, we focus on two objectives. First, the merging process should make good use of all processors, i.e. the workload should be shared. Second, we want to make merging efficient by reducing the number of comparisons between partial skyline points.

Our basic approach for addressing the first objective is rather straight forward. We perform a bottom-up merge, using the recursively constructed space partitioning tree (Algorithm 1). This is clearly advantageous to a centralized merge, as the workload is shared. It also ensures that partial skyline sets merged still have similar projections, so non-skyline points are eliminated early.

We must note though that this basic approach still requires pairwise comparison of all skyline points, which can be expensive if the final skyline set is large. To reduce this cost, which is our second objective, space-partitioning schemes commonly utilize the following two observations:

PROPOSITION 1. *If there exists a coordinate x such that all $p_1 \in P_1$ have lower x-value than all $p_2 \in P_2$, then*

1. *no point in P_2 can dominate a point in P_1, and*
2. *a point in P_1 dominates a point in P_2 iff it dominates it (or is equal to) in the $(d-1)$-dimensional subspace excluding the x-coordinate.*

While the pre-condition of Proposition 1 only holds for the projected points, and points in P_2 can still dominate points in P_1, not all is lost. When comparing partial skyline points $p_1 \in P_1$ and $p_2 \in P_2$, it is still likely (just not certain) that

p_1 has a smaller x-value than p_2. Hence, we will sort partial skyline points in P_2 by their x-value, which enables us to only compare those points in p_1, p_2 where $p_1.x \leq p_2.x$ holds, which likely are few. In our experiments we found that this heuristic works extremely well.

It is also possible to employ this heuristic when filtering out non-skyline points in P_2. Here we sort points in P_1 by the sum of their values *other than* x. We found this heuristic to be beneficial here as well, albeit to a lesser degree, and mainly for cases of low dimensionality.

Furthermore, the second conclusion remains valid, i.e., dominance of points in P_2 by points in P_1 on the $(d-1)$-dimensional subspace implies actual dominance.

THEOREM 2. *Let S_1, S_2 be the subspaces of $\mathbb{R}^d$ defined by*

$$S_1 : \frac{x_1}{x_1 + \ldots + x_d} \leq c$$
$$S_2 : \frac{x_1}{x_1 + \ldots + x_d} > c$$

for some constant c, $p \in S_1$ and $s \in S_2$. Then p dominates s iff p dominates or equals s on $X_2, \ldots, X_d$.

PROOF. The 'only if' direction is trivial, so consider the 'if' direction. By definition of S_1, S_2 we have

$$\begin{aligned} & \frac{p_1}{p_1 + p_2 + \ldots + p_d} \leq c < \frac{s_1}{s_1 + s_2 + \ldots + s_d} \\ \Rightarrow \quad & p_1 \cdot (s_1 + s_2 + \ldots + s_d) < s_1 \cdot (p_1 + p_2 + \ldots + p_d) \\ \Rightarrow \quad & p_1 \cdot (s_2 + \ldots + s_d) < s_1 \cdot (p_2 + \ldots + p_d) \\ & \qquad\qquad\quad \leq s_1 \cdot (s_2 + \ldots + s_d) \\ \Rightarrow \quad & p_1 < s_1 \end{aligned}$$

which shows that p dominates s. □

Hence we can first compute the skyline (approximate or exact) of P_1 w.r.t. $X_2, \ldots, X_d$ before using it to eliminate points in P_2. During our experiments we found that this strategy is useful for up to about 5 dimensions. For more dimensions the number of skyline points in P_1 eliminated becomes too small to compensate for the additional cost in computing the skyline on $X_2, \ldots, X_d$.

A detailed description of our recursive split and merge approach is given in Algorithm 1. Here we use the notations $[p]$ to denote the hyperplane-projection of p, and $p.d$ to denote the d-th coordinate of p. For a set of points P we use the notation

$$[P] := \{[p] \mid p \in P\}$$
$$P.d := \{p.d \mid p \in P\}$$

We note that direct application of Algorithm 1 is only efficient if the entire data set fits into memory. In order to avoid multiple passes for large data sets, we first apply the splitting phase of Algorithm 1 to a small sample set, storing the *split_value*s found in a *space partition tree* (an annotated binary tree describing how the data space is to be partitioned). Afterwards we can use our space partition tree to assign each data point to its partition in a single pass.

3.4 Local Skyline Size

We conclude this section with a brief qualitative analysis regarding the size of local skylines under different partitioning strategies, in order to give the reader a better intuition about them. The central idea is that a "good" partitioning will ensure that many points which dominate each other

Algorithm 1 Split-Skyline-Merge (SSM)

Input: P = points, n = #processors, d = split dimension
Output: skyline of P

if $n = 1$ **then**
  $S := skyline(P)$
  return S
else
  $n_1 := n \ div \ 2$; $n_2 := n - n_1$
  $P_1, P_2 := SPLIT(P, n_1, n_2, d)$
  $next\_d := d + 1 \ mod \ \#dimensions$
  $S_1 := SSM(P_1, n_1, next\_d)$
  $S_2 := SSM(P_2, n_2, next\_d)$
  return $MERGE(S_1, S_2, d)$

Subroutine SPLIT(P, n_1, n_2, d)
Output: P_1, P_2 with $P_1 \cup P_2 = P$, $[P_1].d < [P_2].d$, $\frac{|P_1|}{|P_2|} \approx \frac{n_1}{n_2}$
  $split\_index := |P| \cdot \frac{n_1}{n1+n2}$
  $split\_value := quick\_select([P].d, split\_index)$
  $P_1 := \{p \in P | [p].d \leq split\_value\}$
  $P_2 := \{p \in P | [p].d > split\_value\}$
  return P_1, P_2

Subroutine MERGE(S_1, S_2, d)
Input: local skylines S_1, S_2 with $[S_1].d < [S_2].d$
Output: $skyline(S_1 \cup S_2)$
  sort S_1, S_2 by d-th coordinate
  for all $p_1 \in S_1$ **do**
    for all $p_2 \in S_2$ with $p_2.d \leq p_1.d$ **do**
      if p_2 dominates p_1 **then**
        $S_1 := S_1 \setminus \{p_1\}$
  if #dimensions ≤ 5 **then**
    $S_1' := skyline(S_1)$ on dimensions other than d
  else
    $S_1' := S_1$
  for all $p_2 \in S_2$ **do**
    if $\exists p_1 \in S_1'$ with p_1 dominates p_2 **then**
      $S_2 := S_2 \setminus \{p_2\}$
  return $S_1 \cup S_2$

are assigned to the same partition, so that local skylines are small. In turn, having small local skylines ensures fast response times. In [23] the notion of *pruning power* was introduced, measuring the percentage of space a point p dominates within its partition under some partitioning $\mathcal{X}$. For uniform data distribution, this equates to the probability that it dominates another random point within its partition:

$$\text{Pow}(\mathcal{X}, p) := P\big(p \leq q \mid \mathcal{X}(p) = \mathcal{X}(q)\big)$$

where $\mathcal{X}(p)$ denotes the partition w.r.t. $\mathcal{X}$ in which p lies. A large (average) pruning power typically translates into small local skylines, although the exact relationship between these two measures is complex, even for uniform distributions, and not explored further.

However, it appears that pruning power is ill-suited for qualitative comparison of partitionings, requiring complex analysis even for uniform data distribution [23], thereby providing little intuition. We will thus use the following alternative measure:

DEFINITION 3 (LOCAL MATCH RATIO).
The local match ratio *of a partitioning* $\mathcal{X}$ *is*

$$LMR(\mathcal{X}) := P\big(\mathcal{X}(p) = \mathcal{X}(q) \mid p \leq q\big)$$
$$= P\big(\mathcal{X}(p) = \mathcal{X}(q) \mid p \leq q \vee q \leq p\big)$$

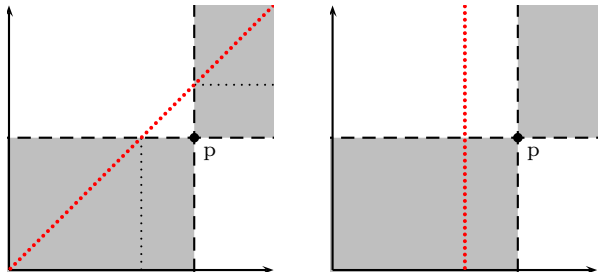
For a fixed point p *the local match ratio w.r.t.* $\mathcal{X}$ *is*

$$LMR(p, \mathcal{X}) := P\big(\mathcal{X}(p) = \mathcal{X}(q) \mid p \leq q \vee q \leq p\big)$$

Note that the average pruning power and the local match ratio of a partition are closely related:

$$\text{LMR}(\mathcal{X}) = \text{Pow}(\mathcal{X}) \cdot \frac{P\big(\mathcal{X}(p) = \mathcal{X}(q)\big)}{P(p \leq q)}$$

Figure 4 illustrates the difference between plane-projection and grid-based partitions for uniform data distribution. Shades areas indicate points comparable to p.



$\text{LMR}(\mathcal{X}) > 0.5$ $\text{LMR}(\mathcal{X}) = 0.5$

Figure 4: Local match ratio for plane-project and grid-based partitioning

From the left graph in Figure 4, it is clear that the LMR for plane-projection based partitioning is at least 0.5 for every point p, resulting in a global LMR value strictly larger than 0.5 (approximately 0.56), i.e., at least half of the shaded area lies in the same partition as p. For grid-based partitioning (right graph in Figure 4), the LMR value is precisely 0.5. An interesting case which highlights the benefits of the LMR measure is random partitioning. Its LMR value is always 0.5 (for 2 equally probable partitions), regardless of dimensionality and data distribution, thus serving as a good baseline for comparison.

While we omit a quantitative analysis (see [23] for details of how pruning power can be calculated - LMR is similar), it should by now be clear that plane-projection based partitioning leads to smaller local skylines than grid-based or random partitioning, at least for uniform data distributions. For anti-correlated data we would expect grid-based partitions to work relatively well. A quantitative comparison for different data distributions and dimensions can be found in the following Section 4.

4. EXPERIMENTS

In order to test the efficiency of our approach for parallel skyline computation, we ran it over randomly generated data sets, and compared it to alternative approaches. Here we considered uniform, correlated and anti-correlated data distributions, under different parameter settings. The experiments were run on a PC with 2.6GHz AMD Athlon 64 X2 dual core processor and 3GB RAM. The default parameter settings are shown in Table 1.

Table 1: Default Parameter Settings

parameter	value	range
nr. of points	100k	10k–1M
dimensions	5	2–8
nr. of processors	8	2–64
network speed	1GBit/s	10MBit/s–∞

For each parameter, we ran one set of experiments (on uniform, correlated and anti-correlated data), where we varied that parameter while keeping the other ones at their default setting. While our implementation is actually sequential, we track time as if computation was performed in parallel, and simulate costs for transporting data between processors (here an infinite network speed means no extra cost, which may be realistic e.g. for multi-core processors, where each core takes the role of a processor).

The different skyline algorithms we considered are

- Plane-Project-Parallel-Skyline (PPPS)
- Angle-based-Parallel-Skyline (APS)
- Grid-based-Parallel-Skyline (GPS)
- Random-Parallel-Skyline (RPS)
- Sort-Filter-Skyline (SFS)

Our algorithm PPPS is described in detail in Section 3. Algorithm APS refers to the Angle-based-space-partitioning approach described in [23], using dynamic space partitioning. The GPS algorithm (Grid-based Parallel skyline) partitions the data set without projecting first, but unlike APS makes use of (a simplified version of) the merge strategy described in Section 3.3. RPS partitions the data set randomly, thus reducing pre-processing costs, at the price of not being able employ efficient merge strategies. The only non-parallel algorithm is SFS [6], which sorts the data set before making dominance checks. SFS is used as basic algorithm for local skyline computation and merging, and for baseline comparison. All skyline algorithms employ the approximate skyline computation pre-processing step described in Section 3.1, as it reduces transport and partitioning costs (this actually makes our SFS implementation more similar to the advanced LESS algorithm [12]).

A direct comparison of computation speed for hypersphere- and hyperplane projection under varying dimensionality is given in Figure 5. As predicted, hyperplane projection is significantly faster, with the difference increasing for growing dimensionality.

Figure 6 shows computation time of the different algorithms for different randomly generated data sets, with varying pairwise correlation. Note that possible correlation values are limited from below by $1/(1-n)$, where n is the number of dimensions (5 here), so for negative correlation the given values are linearly scaled accordingly (-1 indicates pairwise correlation of $-\frac{1}{4}$, $-\frac{1}{2}$ corresponds to pairwise correlation of $-\frac{1}{8}$, etc).

The running times seem to indicate that the benefit of applying parallel algorithms is smaller for uniform or positively correlated data sets. The reason here is that the actual skyline is small, so that a large percentage of the

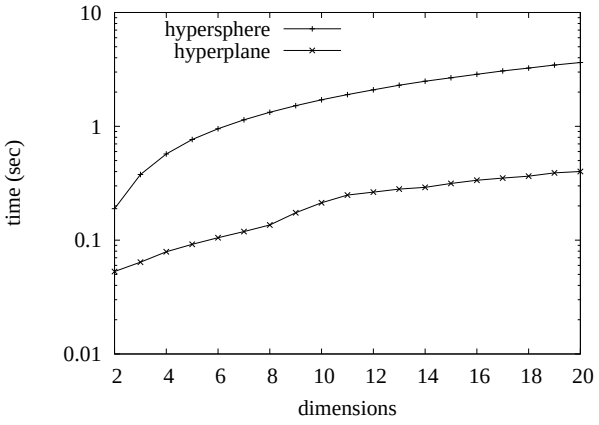


Figure 5: Projection speed comparison

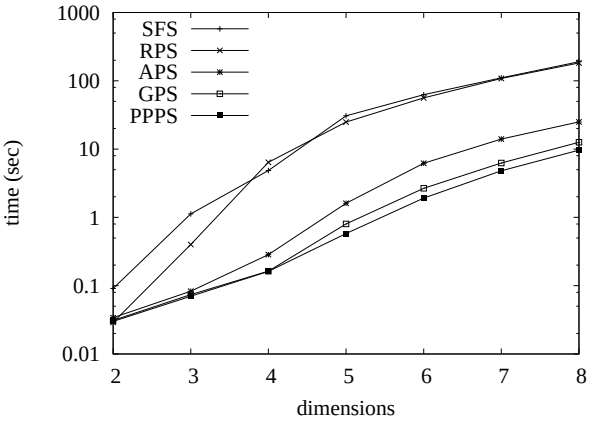


Figure 7: Varying dimensions (anti-correlated)

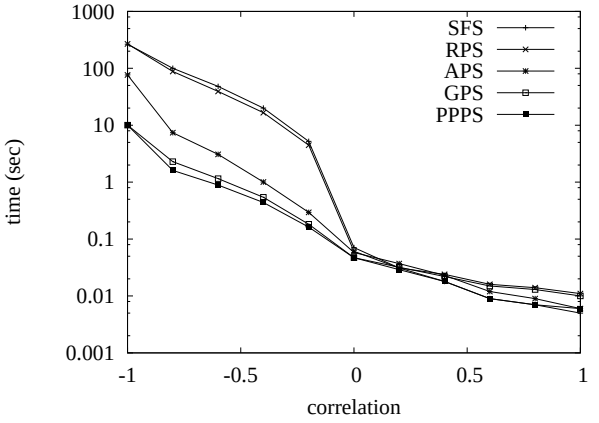


Figure 6: Varying correlation

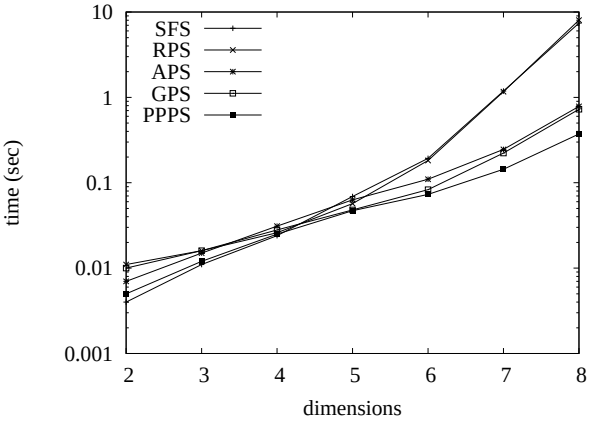


Figure 8: Varying dimensions (uniform)

time is just spent on pre-processing steps and data transport. However, larger skylines can also occur for uniform and positively correlated data when the dimensionality is high. Figures 7, 8 and 9 show results for varying number of dimensions and anti-correlated (corr=-0.5), uniform and correlated (corr=0.5) data, respectively.

Once again we see that parallel skyline computation is only effective in cases where the actual skyline is reasonably large, this time due to high dimensionality. In all such cases, our PPPS algorithm is considerably faster (note that time is in log-scale) than the alternatives considered. It is worth noting that the difference to the angle-based partitioning approach (APS) is not just due to the faster projection method, but mainly caused by the efficient merge algorithm. GPS outperforms APS for the same reason.

In order to judge the impact of the different optimization applied, namely

1. approximate skyline pre-processing, and
2. efficient merging

we compare our PPPS algorithm with variants employing none or only one of the above optimization. Figure 10 shows the behavior of the different algorithms for varying correlations. PPPS(A+M) denotes the PPPS algorithm with both approximate skyline pre-processing and efficient merge, PPPS(M) is PPPS without approximate skyline, etc.

We find that approximate skyline pre-processing is helpful for uniform or correlated data sets, as it reduces transport costs as well as sorting costs for SFS. For anti-correlated data is has virtually no impact, as other costs dominate and the approximate skyline grows in size. Efficient merging on the other hand is extremely beneficial for anti-correlated data, but actually increases processing costs for uniform and correlated data sets, due to overheads exceeding costs for simple "brute-force" skyline merge.

Figures 11, 12 and 13 show how the different algorithms scale with data set size. The dimensionality used is 5, 10 and 15 for the anti-correlated, uniform and correlated case respectively, to ensure that the skyline is large enough to necessitate parallel skyline algorithms.

As is to be expected, the increase in time is more than linear but slightly less than quadratic in the size increase, similar to the underlying SFS algorithm used for local skyline computation. The difference in speed between the algorithms grows slightly with increasing data set size, up to about 50,000, from which point on it remains fairly constant. This is due to the preprocessing costs (approximate skyline computation, partitioning, transport costs) becoming less significant, as they grow only linearly in the data size. Our

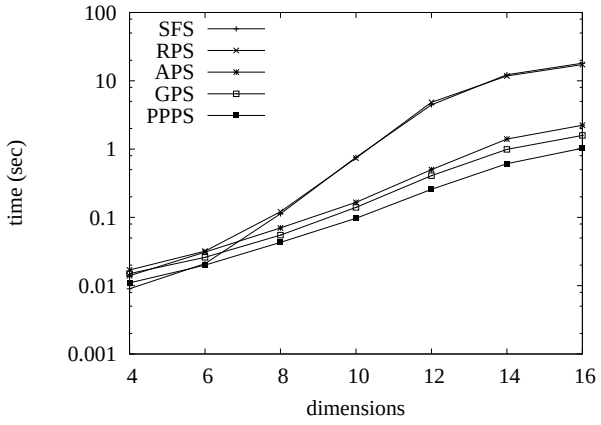


Figure 9: Varying dimensions (correlated)

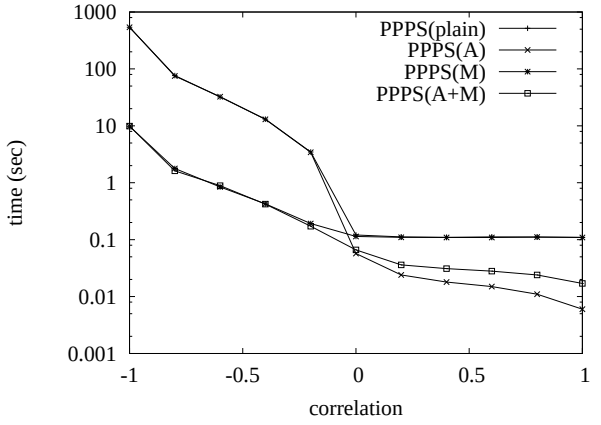


Figure 10: Varying correlation (by optimization)

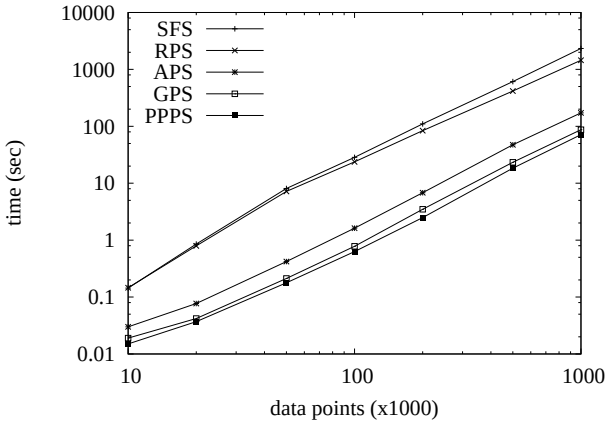


Figure 11: Varying size (anti-correlated)

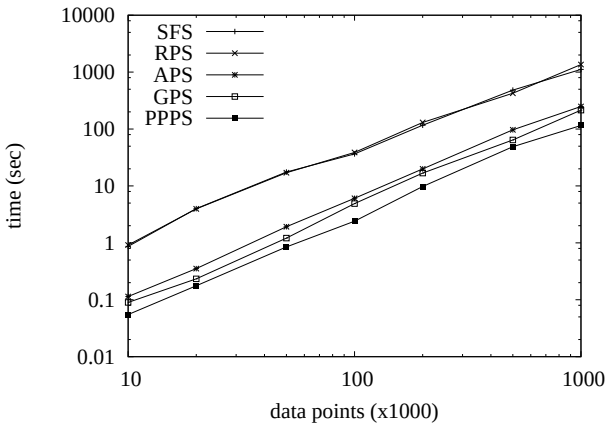


Figure 12: Varying size (uniform)

PPPS algorithm is consistently the fastest, by more than one order of magnitude compared to SFS. This is interesting in so far as the speed factor exceeds the number of processors (8 here). The explanation is that the efficient merge step helps greatly in reducing skyline point comparisons, making our approach superior to SFS/LESS even when run sequentially.

The effect that the number of processors has on processing time is shown in Figures 14, 15 and 16.

For PPPS, computation time drops quickly with rising number of processors, up to about 32 for our anti-correlated data set, at which point preprocessing and merge costs become dominant and further increases in the number of processors bring no significant benefit. For APS this point is reached earlier at about 8 processors, and later for GPS, while the random partitioning approach (RPS) hardly profits from parallel processing power at all. Again, the reason is an inefficient merge step, which becomes more costly as the partitions become finer. We conclude that random partitioning is not suitable for parallel skyline computation, regardless of the number of processors available. We found that the point at which further increases in the number of processors does not improve performance of PPPS (respectively APS) increases slowly with growing data set size.

It is worth noting that in cases where the skyline set is large, merging of local skylines can be quite expensive. On the other hand, this step often eliminates only a very small fraction of points. Thus, if a good approximation of the actual skyline set is sufficient for the application scenario, one may want to omit this final merge step. This is particularly profitable if the application requires further parallel processing of the skyline set, as this may eliminate the need for synchronization and merge completely. Table 2 shows the size of the actual skyline and of the (union of) local skyline sets resulting from the different partitioning approaches, for different data sets.

Table 2: Local-Approximate Skyline Sizes

dimensions	5	10	15
correlation	anti-corr	uniform	correlated
skyline size	17668	26442	13812
plane-project	18378 (+4%)	34366 (+30%)	17646 (+28%)
angle-based	18363 (+4%)	34884 (+32%)	17962 (+30%)
grid-based	19578 (+11%)	46568 (+76%)	20910 (+51%)
random	26783 (+52%)	47584 (+80%)	20618 (+49%)

We find that random partitioning leads to large local skylines in all cases. Grid-based partitioning provides small local skylines for anti-correlated data in few dimensions, albeit still larger than for plane-project or angle-based pro-

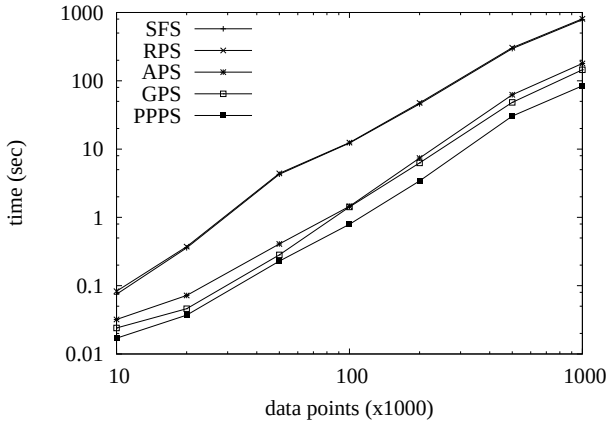


Figure 13: Varying size (correlated)

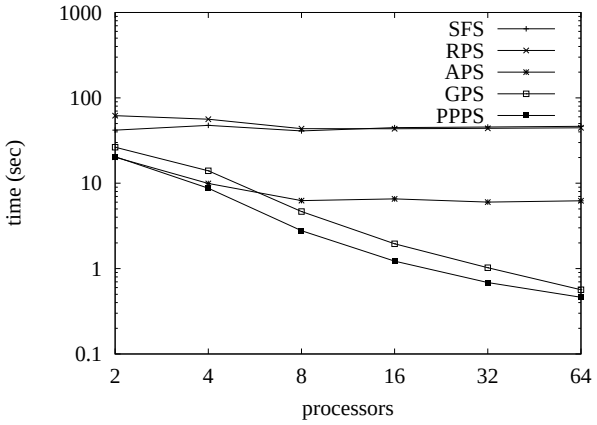


Figure 15: Number of processors (uniform)

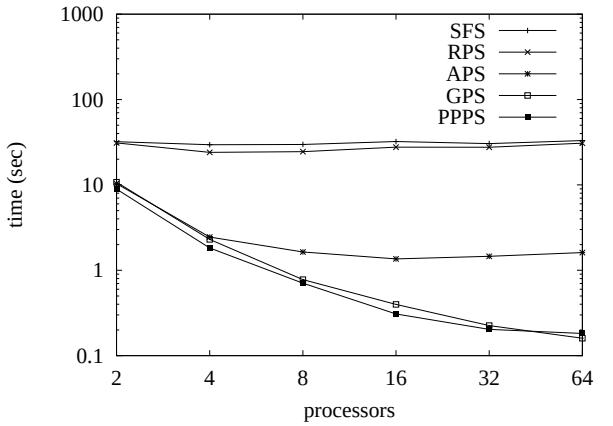


Figure 14: Number of processors (anti-correlated)

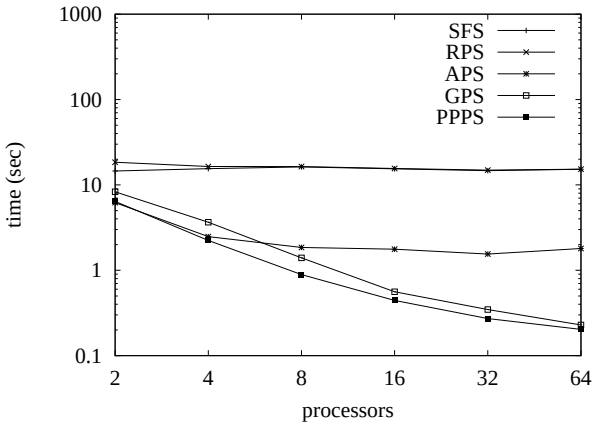


Figure 16: Number of processors (correlated)

jection, but fails for uniform or correlated ones with high dimensionality. This confirms the quantitative results of Section 3.4. Plane-projection and angle-based partitioning approaches perform best in all cases, but the difference between local skyline and global skyline is still significant for high-dimensional data.

5. CONCLUSION

We have investigated a novel approach to parallel skyline computation, based on a data partitioning strategy which uses projection of points onto a hyperplane. It combines the benefits of small local skylines, as achieved by angle-based partitioning [23], with an effective merge strategy, as used in various grid-based partitioning schemes, and outperforms both. In addition, we investigated the benefits of initial approximate skyline pre-computation, and the impact of various optimizations under different data distributions. For future work, we will to investigate a combination of our partitioning strategy with the recursive merge-approach devised in [15].

Acknowledgement: The work reported in this paper is supported by the Australian Research Council research grants DP0987557 and LP0882957. Yang's research is also partially supported by the National Science Foundation of China (grant number: 61070056, 61033010) and National High Technology Research and Development Program 863 of China (grant number: 2008AA01Z120).

6. REFERENCES

[1] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *EDBT*, pages 256–273, 2004.

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

[3] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

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

[5] M. J. Carey and D. Kossmann. On saying 'Enough Already!' in SQL. In *SIGMOD*, pages 219–230, 1997.

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

[7] A. Cosgaya-Lozano, A. Rau-Chaplin, and N. Zeh. Parallel computation of skyline queries. In *21st International Symposium on High Performance Computing Systems and Applications (HPCS)*, page 12, 2007.

[8] B. Cui, H. Lu, Q. Xu, L. Chen, Y. Dai, and Y. Zhou. Parallel distributed processing of constrained skyline queries by filtering. In *ICDE*, pages 546–555, 2008.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.

[10] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *9th Annual Symposium on Computational Geometry (SCG)*, pages 298–307, 1993.

[11] Y. Gao, G. Chen, L. Chen, and C. Chen. Parallelizing progressive computation for skyline queries in multi-disk environment. In *DEXA*, pages 697–706, 2006.

[12] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, pages 229–240, 2005.

[13] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in manets. In *ICDE*, page 66, 2006.

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

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

[16] J. Matoušek. Computing dominances in e^n (short communication). *Information Processing Letters*, 38(5):277–278, 1991.

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

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

[19] S. Park, T. Kim, J. Park, J. Kim, and H. Im. Parallel skyline computation on multicore architectures. In *ICDE*, pages 760–771, 2009.

[20] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, 2007.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

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

[23] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *SIGMOD*, pages 227–238, 2008.

[24] S. Wang, B. C. Ooi, and A. K. H. Tung. Efficient skyline query processing on peer-to-peer networks. In *ICDE*, pages 1126–1135, 2007.

[25] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *EDBT*, pages 112–130, 2006.

[26] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, 2007.