Universität Augsburg

Parallel Multi-Level Preference Computation

M. Endres, S. Wohlfart

Report 2017-03 July 3, 2017

Institut für Informatik
D-86135 Augsburg

Parallel Multi-Level Preference Computation

Markus Endres, Stefan Wohlfart

Institut für Informatik, Universität Augsburg, D-86135 Augsburg, Germany
{markus.endres, stefan.wohlfart}@informatik.uni-augsburg.de

Abstract. Given a data set, a *top-k Skyline* query returns the k *most interesting* elements of the Skyline query based on some kind of user-defined preference. That means, sometimes not only the Pareto frontier is of interest, but also the stratum, the level, behind the Skyline to get exactly the *top-k* objects from a partially ordered set stratified into subsets of non-dominated tuples. In this paper, we extend the definition of top-k Skyline to form *multi-level Skyline* sets. Multi-level Skylines are a variant of top-k Skylines which do not stop after k tuples, but compute all Skyline levels. We present a parallel algorithm for multi-level Skyline computation on multi-core architectures and demonstrate through extensive experimentation on synthetic and real data sets that our algorithms can result in a significant performance advantage over existing techniques.

Keywords: Preference, Multi-level preference, Top-k, Skyline, Multi-core, Parallel computation

1 Introduction

Skyline and Pareto preference queries [BKS01,CCM13,Kie02,KEW11,DPE08] are well known in the database and artificial intelligence community. The Skyline contains the objects that are not dominated by any others based on a user's defined preference. However, the drawback of this approach is that the output size cannot be controlled. As a result the output can contain too few or too many objects. While too few objects can result in the user choosing none, too many make it harder for the user to make any decision. Therefore, top-k and multi-level Skylines were introduced [EP15,PE15]. This approach selects only the k best objects and further retrieves the objects directly dominated by those of the Skyline if more are needed. Hence, the multi-level Skyline consists of levels, each containing a different "Skyline". Skyline objects will only be dominated by objects in lower levels.

Example 1.1. Let us search for a hotel that is *cheap* and *close to the beach*. Unfortunately, these two goals are conflicting as the hotels near the beach tend to be more expensive. Interesting are all hotels that are not worse than any other hotel in these both dimensions. Table 1 presents a sample data set.

Table 1: Sample data set of hotels.

hotel	id	beach dist. (km)	price (€)	board
	p1	2.00	25	none
	p2	1.25	50	breakfast
	p3	0.75	75	half board
	p4	0.50	150	full board
	p5	0.25	225	full board
	p6	1.75	110	half board
	p7	1.10	120	breakfast
	p8	0.75	220	full board
	p9	1.60	165	half board
	p10	1.50	185	breakfast

The hotels p_6, p_7, p_9, p_{10} are dominated by hotel p_3. The hotel p_8 is dominated by p_4, while the hotels p_1, p_2, p_3, p_4, p_5 are not dominated by any other hotels and build the *Skyline* $\mathcal{S}$, cp. Figure 1.

In our example above maybe five hotels are not enough, so we have to present the next Skyline level called $\mathcal{S}^1_{ml}$ (*Skyline, multi-level 1*, dashed line in Figure 1): p_6, p_7, p_8. Also, the third best result set $\mathcal{S}^2_{ml}$ might be of interest: p_9, p_{10}.

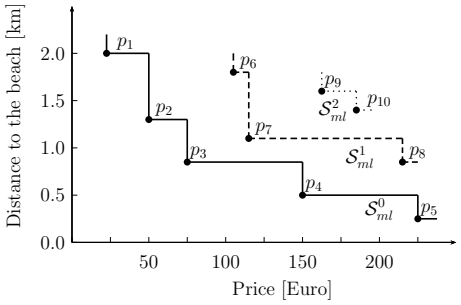


Fig. 1: Multi-level Skylines.

In [EP15,PE15] we presented how to evaluate such multi-level Skylines exploiting the lattice structure constructed by a Skyline query over low-cardinality domains. However, many Skyline applications involve domains with high cardinalities, such that computing the *l-th Skyline level* must be processed efficiently. In this technical report we propose an efficient strategy for multi-level Skylines computation on multi-core architectures. Our algorithm consists of two parallel and one sequential phase and shows its advantage to other existing approaches.

The remainder of the paper is organized as follows: Section 2 presents related work and their approaches for the mentioned problem. Section 3 defines the Skyline, the multi-level Skyline and top-k Skylines. Section 4 explains the basics of the parallel algorithm based on a sequential variant. Section 5 provides the details and a pseudo code for our parallel algorithm. Section 6 shows comprehensive experimental results and how the algorithm copes with existing approaches. The last Section, Section 7, provides a conclusion and future work on how to improve the parallel version.

2 Related Work

A Skyline or Pareto preference result set can be identified for example with one of the following algorithms: Block-Nested-Loop (BNL) [BKS01], Divide & Conquer [BKS01], Sort-Filter-Skyline (SFS) [CGGL03] or LESS (Linear Elimination Sort for Skyline) [GSG07]. Many of these algorithms have been adapted for parallel Skyline computation, e.g., [SLB10,LVDN,CSAB15]. There are also algorithms utilizing an index structure to compute the Skyline, e.g., [PTFS03,LZLL07,EW17], just to name a few. Another approach exploits the lattice structure induced by a Skyline query over low-cardinality domains. Instead of direct comparisons of tuples, a lattice structure represents the better-than relationships. Examples for such algorithms are *Lattice Skyline* [MPJ07] and *Hexagon* [PK07], both having a *linear time complexity*. There is also work on parallel preference computation exploiting the lattice structure [EK14,EK15]. The authors of [ERK15] present how to handle high-cardinality domains and therefore makes lattice algorithms available for a broad scope of applications, see also [EP17]. Further work and variants on Skylines are, e.g., [End15,EK11,ERK14].

Due to the fact that the output size cannot be controlled in the above mentioned algorithms, many approaches were invented to return the best k objects, e.g., [BG04,GV05a,LYLC06,BGG07], or to cluster Pareto optimal results [KEK17]. They combined the Skyline and top-k approach to reduce the result to k tuples. This can be done by computing the Skyline and applying some sort of post-processing afterwards. In [GV05b] and [BGG07]

the authors calculate the first stratum/level of the *Skyline* with some sort of post-processing. Afterward, they define the k best objects or continue Skyline computation without the first level. The authors of [LYH09] abstract Skyline ranking as a dynamic search over Skyline subspaces guided by user-specific preferences.

In [TXP07,GV09] and [PP09] an index based approach is used for top-k Skyline computation. However, index based algorithms in general cannot be used if there is a join or Cartesian product involved in the query. Su et al. [SCL10] consider top-k combinatorial Skyline queries, and Zhang et al. [ZZL+11] discuss a probabilistic top-k Skyline operator over uncertain data. Top-k queries are also of interest in the computation of spatial preferences [YDMV07,RJVDN10], where the aim is to retrieve the k best objects in a spatial neighborhood of a feature object. Yu et al. [YAY12] consider the problem of processing a large number of continuous top-k queries, each with its own preference. The authors of [VDNV08] present a framework for top-k query processing in large-scale P2P networks, where the data set is horizontally distributed to peers. For this they compute k-skyband sets as a pre-processing step, which are aggregated to answer any incoming top-k queries.

Another approach which addresses the top-k problem is to use multi-Level preference queries [EP15,PE15], where all levels "behind" the Skyline are computed to solve the top-k approach. The idea of *multi-level preferences* was already mentioned by Chomicki [Cho03] under the name of *iterated preferences*. However, Chomicki has never presented an algorithm for the computation of multi-level preferences. The algorithms presented in [EP15,PE15] rely on the lattice structure constructed by a Pareto preference query over low-cardinality and support multi-level preference queries as well as top-k Skylines. However, this approach cannot exploit modern multi-core architectures. Thus, our report will present a parallel algorithm which not only supports multi-level preference query evaluation, but is also more efficient for high dimensional data sets than previous attempts.

3 Preliminaries

A Pareto preference or Skyline query selects those tuples from a data set that are optimal with respect to a user defined multicriteria function. Hence, a Skyline is a subset where every tuple is not dominated by any other tuple in the original data set. A tuple dominates another one if it is better in one dimension and at least equally good in all other dimensions. Thus, the dominance is defined as follows:

Definition 3.1 (Dominance and Indifference). *Assume a set of vectors $D \in \mathbb{R}^d$. The so called Pareto ordering ($\otimes$) is defined for all $x = (x_1, ..., x_d)$, $y = (y_1, ..., y_d) \in D$ with d dimensions:*

$$x <_\otimes y \iff \begin{array}{l} \forall j \in \{1, ..., d\} : x_j \leq y_j \ \wedge \\ \exists i \in \{1, ..., d\} : x_i < y_i \end{array} \tag{1}$$

x and y are called indifferent on D, denoted as $x \sim y$ if and only if $\neg(x <_\otimes y) \wedge \neg(y <_\otimes x)$.

On basis of the dominance the Skyline can be defined as follows:

Definition 3.2 (Skyline). *The* Skyline *$\mathcal{S}$ of D is defined by the maxima in D according to the ordering $<_\otimes$, or explicitly by the set*

$$\mathcal{S}(D) = \{t \in D \mid \nexists u \in D : u <_\otimes t\} \tag{2}$$

In this sense the minimal values in each domain are preferred and are written as $x <_\otimes y$ if x is better than y.

If more tuples are needed than available in the Skyline, succeeding multi-level Skylines need to be considered. These Skylines will be determined in the same way as the original one but without considering tuples from previous Skyline sets. Each Skyline will be added to a level starting at level 0. Thus, the multi-level Skyline can be retrieved by assigning every tuple to its appropriate level.

Definition 3.3 (Multi-Level Skyline [EP15]). *The multi-level Skyline set of level l (i.e., the l-th stratum) for a data set D is defined as*

$$\mathcal{S}_{ml}^{l} := \mathcal{S}\left(D \setminus \bigcup_{i=0}^{l-1} \mathcal{S}_{ml}^{i}(D)\right) \tag{3}$$

Thereby $\mathcal{S}_{ml}^{0}(D)$ is identical to the standard Skyline $\mathcal{S}(D)$, and $\mathcal{S}_{ml}^{l_{max}}$ denotes the non-empty set with the highest level.

If there aren't enough tuple in level 0, the next level of the multi-level Skyline needs to be computed to specify the best k tuples.

Definition 3.4 (Top-k Skyline). *A* top-k Skyline *query $\mathcal{S}_{tk}^{k}(D)$ on an input data set D computes the top k elements with respect to the Skyline preferences. Formally:*

1. *If $|\mathcal{S}(D)| > k$, then return only k tuples from $\mathcal{S}(D)$, because not all elements can be returned due to result set size limitations. Any k tuples are a correct choice.*
2. *If $|\mathcal{S}(D)| = k$, then $\mathcal{S}_{tk}^{k}(D) = \mathcal{S}(D)$. That means return all tuples of $\mathcal{S}_{ml}^{0}(D)$. In this case there is no difference between the* Skyline *set and the top-k result set.*
3. *If $|\mathcal{S}(D)| < k$, then the elements of $S(D)$ are not enough for an adequate answer. We have to find a value j which meets the following criterion:*

$$\left|\bigcup_{i=0}^{j-1} \mathcal{S}_{ml}^{i}(D)\right| < k \leq \left|\bigcup_{i=0}^{j} \mathcal{S}_{ml}^{i}(D)\right| \tag{4}$$

That means, not only all elements of $\mathcal{S}(D) = \mathcal{S}_{ml}^{0}(D)$ are returned, but also some of $\mathcal{S}_{ml}^{1}(D)$, and if the number of result tuples is still less than k, then $\mathcal{S}_{ml}^{2}(D)$, and so on. Note that from $\mathcal{S}_{ml}^{j}(D)$ exactly $k - \left|\bigcup_{i=0}^{j-1} \mathcal{S}_{ml}^{i}(D)\right|$ elements will be returned, which might not be all of it.

4 Sequential Multi-Level Skyline Algorithm

Before we explain our parallel algorithm, a sequential variant is introduced to explain the overall idea of the algorithm. Our algorithms are extensions of the multi-core Skyline algorithm presented in [CSAB15] with adaptations for multi-level Skyline evaluation.

Our sequential approach is depicted in Algorithm 1. First of all the data set will be sorted by the Manhattan norm, also called L_1. This ensures that no succeeding tuples in the data set dominates preceding objects. However, it can happen that following tuples are indifferent to all tuples in a prior level and therefore will be placed in earlier levels. Due to this reason the algorithm can only be stopped, without computing the whole multi-level Skyline, if the first level has k points. This can be checked after each tuple iteration (after line 22). After sorting, each tuple in the data set will be compared to each level and tuple of the multi-level Skyline. If the point is not dominated by all the points of a level, it will be added to this specific level. Otherwise, it will be compared to all following levels until one is found which contains no points which dominate it.

Given the data set in Table 2 (left) the sequential algorithm generates the output next to it (right table). The left column of the right table corresponds to the different levels of the multi-level Skyline ($\mathcal{S}_{ml}$), whereas the right column contains the ID of the tuples assigned to this level.

Algorithm 1 Sequential Multi-Level Skyline Algorithm based on L_1

Input: Data set D, top-k value k
Output: k Multi-level Skyline sets w.r.t. MIN preference

```
1: Sort D according to L_1 := sum_{i=1}^{d} x_i, x_i ∈ D
2: S_ml[k] ← [<list>]                          // initialize array (size k) of lists to store S_ml sets
3:                                             // S_ml[0] is the first level, i.e., S(D)
4: level ← 0                                   // Maximum index of known Skyline level
5: for all p ∈ D do
6:     for i = 0 to level do
7:         isSkylinePoint ← TRUE
8:         for all q ∈ S_ml[i] do              // Check each point in the i-th level
9:             if q <_⊗ p then                 // q dominates p
10:                isSkylinePoint ← FALSE      // p is dominated by any q in S_ml[i]
11:                break 'for all' loop line 8 // Goto line 14
12:            end if
13:        end for
14:        if isSkylinePoint then
15:            S_ml[i].add(p)                  // Add p to the i-th level
16:            break 'for all' loop line 6     // Goto line 19
17:        end if
18:    end for
19:    if !isSkylinePoint and level < k − 1 then   // k multi-levels, 0, ··· , k − 1
20:        level++
21:        S_ml[level].add(p)
22:    end if
23: end for
24: return the first k elements from S_ml
```

ID	Distance (meter)	Price (€)	L_1
1	100	40	140
2	150	200	350
3	300	140	440
4	200	280	480
5	400	100	500
6	350	220	570
7	500	120	620
8	600	160	760
9	550	240	790
10	600	320	920

S_{ml}	ID
S_{ml0}	{1}
S_{ml1}	{2,3,5}
S_{ml2}	{4,6,7]}
S_{ml3}	{8,9}
S_{ml4}	{10}

Table 2: Sequential algorithm example

5 Parallel Multi-Level Skyline Algorithm

This section provides details on our parallel algorithm which is based on the sequential approach presented in Section 4. The algorithm is shown in Algorithm 2. Again, we sort the data by the Manhattan norm (line 1). Afterwards the data will be divided into b blocks. Each block will be processed successively in two parallel and one sequential phase. The first phase (`CompareToSml`) compares each tuple of the block with the current multi-level Skyline. In the second phase (`CompareToPeers`) each tuple will be compared to all preceding tuples of the same block. In the final sequential step (`UpdateGlobalSkylineSml`) each tuple of a block will be added, based on the results of the previous phases, to the appropriate level of the multi-level Skyline. After every block was processed, the first k elements starting from level 0 will be returned.

Algorithm 2 Parallel Multi-Level Algorithm

Input: Data set D, k, block size b
Output: k Multi-Level Skyline sets w.r.t. MIN preference

```
1: Sort D according to L_1 := sum_{i=1}^{d} x_i, x_i ∈ D
2: S_ml[k] ← [<list>]                                   // initialize global multi-levels S_ml
3: while D ≠ ∅ do
4:     B ← next b points from D                          // Read data in blocks
5:     D ← D \ B
6:     B = CompareToSml(S_ml, B)                         // Parallel Phase I
7:     B = CompareToPeers(B)                             // Parallel Phase II
8:     UpdateGlobalSkylineSml(S_ml, B)                   // Phase III
9: end while
10: return the first k elements from S_ml
```

During the `CompareToSml` phase (see Algorithm 3) a tuple will be compared to all levels and all tuples in the multi-level Skyline. If it is not dominated by all tuples of one of the levels, this level will be assigned to the tuple (line 11). Otherwise, it will be marked as pruned and won't be considered anymore.

Algorithm 3 CompareToSml

Input: $\mathcal{S}_{ml}$, B
Output: B

```
1: parallel for each b ∈ B do
2:     for j = 0 to k do
3:         mark b as not dominated
4:         for all q in level S_ml[j] do
5:             if q <_⊗ b then
6:                 mark b as dominated
7:                 break 'for loop' line 4                  // Goto line 10
8:             end if
9:         end for
10:        if b is not marked as dominated then
11:            b.level ← j                                  // b is not dominated by any q in S_ml[j]
12:            break 'for loop' line 2                      // Goto line 15
13:        end if
14:    end for
15:    if b is marked as dominated then                     // all k multi-levels are done
16:        mark b as pruned
17:    end if
18: end parallel for each
19: return B
```

The second phase `CompareToPeers` (see Algorithm 4) compares all tuples in the block to its predecessors. If one of them dominates the tuple, it will be added to a list of dominator (line 9). Only predecessors need to be considered because due to the sorting no following tuple in the same block will dominate it.

Unlike the other phases `UpdateGlobalSkylineSml` (see Algorithm 5) runs sequential and adds each tuple of a block to its respective level. There are two cases which needs to be considered to assign a tuple its correct level. Firstly, the level needs to be higher than the level of all tuples in the multi-level Skyline which dominate it. This level was assigned to the tuple as a result of the `CompareToSml` phase. Secondly, the level should be higher than the level of every tuple in the same block dominating it. The levels of the dominator can be retrieved from the dominator list computed in the `CompareToPeers` phase.

With this list, the maximum level from the dominator can be identified (line 6). If the tuple's level is smaller than this maximum level plus one, maximum level plus one is its new level. Afterwards it will be checked if its level is smaller than k. In this case it will be added to this level of the multi-level Skyline. Otherwise, the tuple will be discarded. For an more efficient runtime it can be checked if level 0 contains already k tuples (after line 8 of Algorithm 2). However, this only works with the lowest level because it can happen that later tuples still will be added to this level. Thus, returning tuples from the next lowest level before computing the whole multi-level Skyline, can falsify the results.

Algorithm 4 CompareToPeers

Input: B
Output: B

```
 1: parallel for each b ∈ B do
 2:     if b is not marked as pruned then
 3:         for j = 1 to b.index do          // index corresponds to the position w.r.t. L1 norm, i.e., B[i]
 4:             if B[j] <⊗ b then                                   // b = B[i] at position i
 5:                 if B[j].level == k − 1 then                     // k multi-levels from 0 to k − 1
 6:                     mark b as pruned                // b is worse than any predecessor in level k − 1
 7:                     break 'for loop' line 3                                  // Goto line 13
 8:                 else
 9:                     b.dominator.add(B[j])                                    // B[j] dominates b
10:                 end if
11:             end if
12:         end for
13:     end if
14: end parallel for each
15: return B
```

Algorithm 5 UpdateGlobalSkylineSml

Input: $\mathcal{S}_{ml}$, B

```
 1: for i = 0 to b do
 2:     if B[i] is not marked as pruned then
 3:         if B[i].dominator.size() != 0 then                          // There are dominator
 4:             maxlevel = max{q.level | q ∈ B[i].dominator}
 5:             if B[i].level < maxlevel + 1 then
 6:                 B.level ← maxlevel + 1
 7:             end if
 8:         end if
 9:         if B[i].level < k then                                              // 0 . . . k − 1
10:             S_ml[B[i].level].add(B[i])
11:         end if
12:     end if
13: end for
```

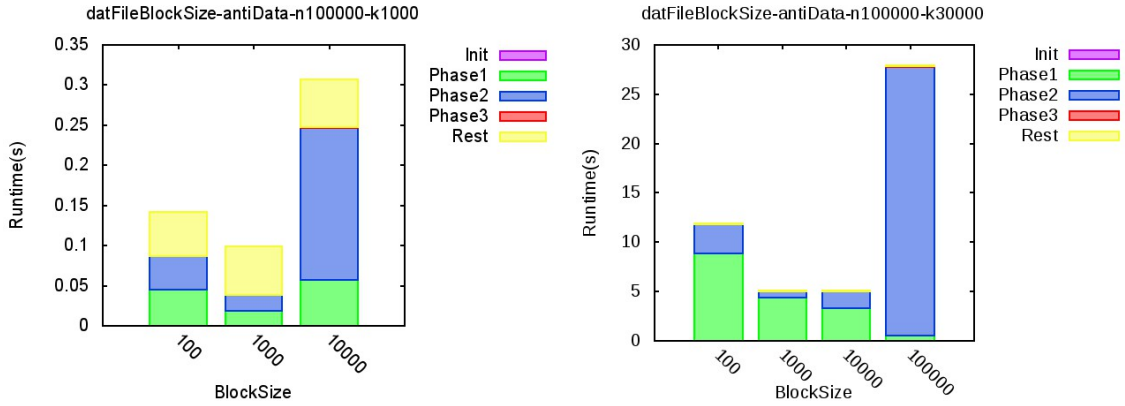
6 Experiments

In this section we compare our parallel multi-level algorithm with two already existing approaches: EBNL and ESFS [BGG07]. For the EBNL every tuple will be compared to a window which keeps all non-dominated tuples. After comparing every tuple to the window, removing all dominated tuples and adding all non-dominated tuples, the window only holds the best tuples which form the Skyline. Thereafter, the window will be cleared and the remaining tuples will be processed in the same fashion yielding the next Skyline. This process terminates if the size of all computed Skylines in sum is higher or equal to k. ESFS on the other hand first sorts the data due to the entropy function $E(t) = \sum_{i=1}^{d} \ln(x_i + 1)$. Afterwards it compares the tuples to the window in the same fashion as EBNL. However, due to the sorting a tuple doesn't need to be checked if it dominates a tuple from the window anymore (cp. [BGG07]).

Our experiments were performed on a server running Debian GNU/Linux 7.11 with 44 GB of free RAM. Furthermore, it is equipped with an Intel(R) Xeon(R) E5540 with 16 cores each with 2,53 GHz and a cache size of 8 MB. The parallel algorithm was tested with different block, thread and input sizes. In addition, all algorithms were compared by varying the input parameters k and the number of dimensions. To enable experiments with different input sizes, a data generator as mentioned in [BKS01] was developed. It generates correlated, anti-correlated and independent data with different input parameters. Moreover, benchmark tests on real data were realized. For this the free data sets NBA, House and Zillow were used.

6.1 Influence of the Blocksize

We analyze the behavior of the block size to assist in selecting the most efficient size for a specific input data set. Figure 2 shows two benchmark tests with different top-k values on an anti correlated data which consists of 100.000 tuples and 5 dimensions. Figure 2(a) has its most efficient block size at 1.000, whereas Figure 2(b) has its own at 10.000. The different runtimes and block sizes can be explained by the value of k. While Figure 2(a) has a low value, Figure 2(b) has a high value for k exceeding the size of the first level of the multi-level Skyline. Therefore, the algorithm will never find k tuples in level 0 and thus needs to determine further multi-level Skyline sets which causes a higher runtime.

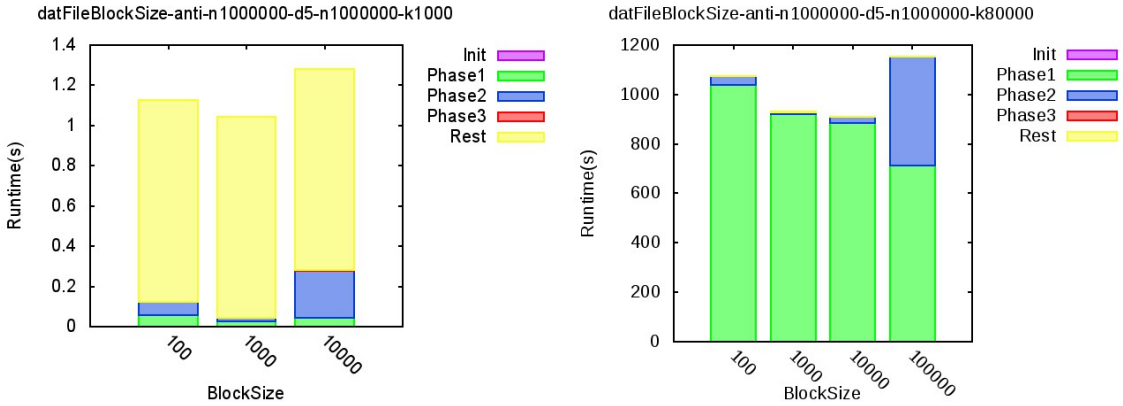


(a) Top-k: 1.000

(b) Top-k: 30.000

Fig. 2: Block size - n: 100.000

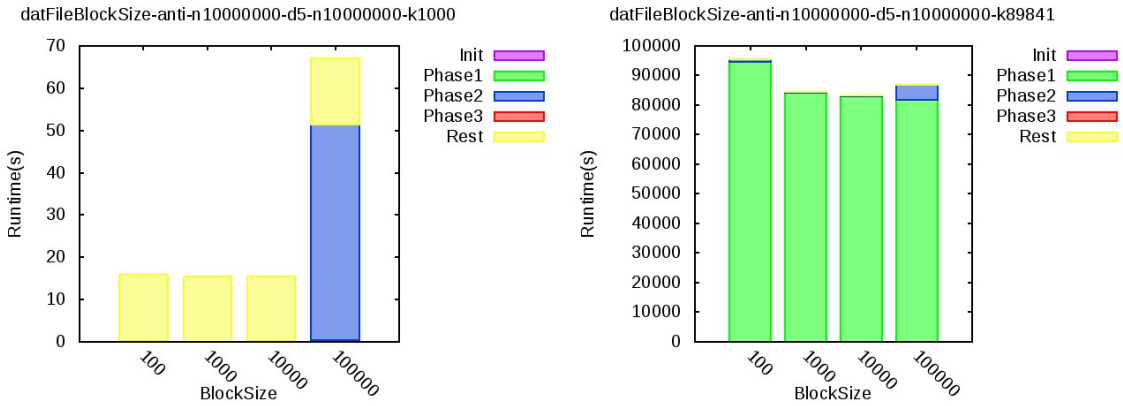
For an input size of 1.000.000 objects the best performing block size is the same. Figure 3 shows that a higher block size leads to a longer phase 2 and a faster phase 1. This is intuitive since the data will be divided into less blocks resulting in less comparisons in phase 1 and more in phase 2. The more tuples need to be compared in phase 2 the larger the dominator lists. Since these lists are kept in memory, it can be possible that they exceed the available memory. Therefore, block sizes which exceed 100.000 or more are very inefficient.



(a) Top-k: 1.000 (b) Top-k: 80.000

Fig. 3: Block size - n: 1.000.000

These findings can also be seen in Figure 4, in which the block size benchmark was applied to a data set with 10.000.000 tuples. In summary, the smaller k and the input size, the smaller the block size. Due to the limit of the available memory higher k and input sizes result in an optimal block size of 10.000.

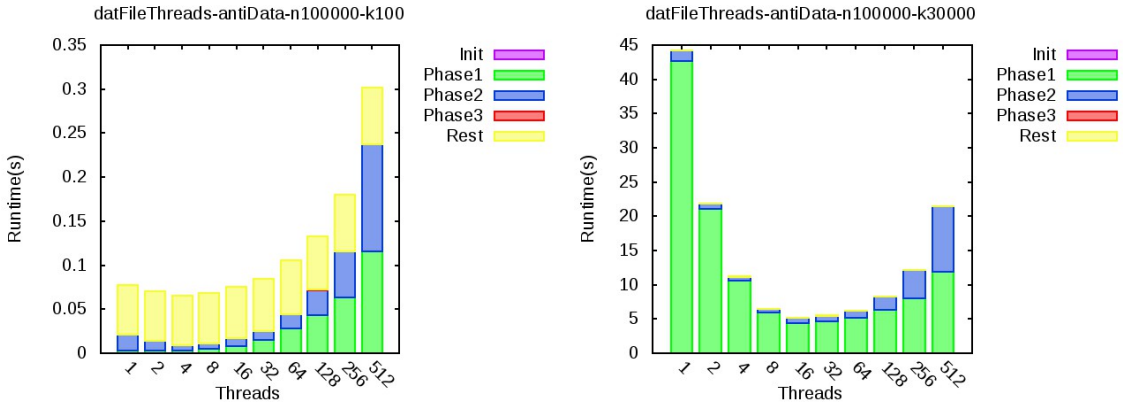


(a) Top-k: 1.000 (b) Top-k: 89.841

Fig. 4: Block size - n: 10.000.000

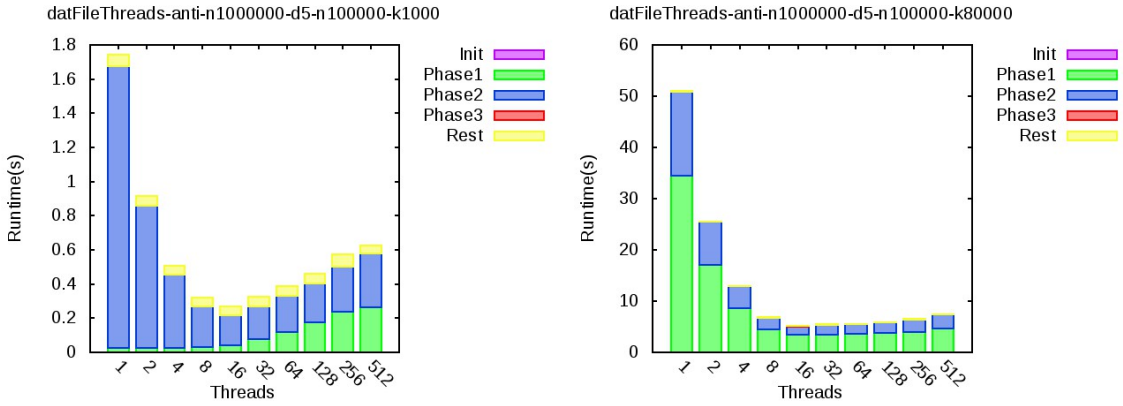
6.2 Influence of the Number of Threads

This section discusses how the thread size and different k values affect the runtime of the parallel algorithm. Figure 5(a) shows that for a input size of 100.000 and a low k, a thread size of 4 is optimal. For a k which is higher than the number of tuples in level 0, the optimal size of the threads equals the number of CPU cores, cp. Figure 5(b). It can also be noticed that a higher number of threads than the number of cores increases the runtime. The cause for this is the overhead of the threads since threads need to be created and need to wait for others to finish. Therefore, a trade-off between parallelism and the overhead need to be inspected, to select the optimal thread size. In Figure 6 we see that for an input size of 1.000.000 the optimal thread size equals to the numbers of cores available.



(a) Top-k: 1.000 (b) Top-k: 30.000

Fig. 5: Thread size - n: 100.000



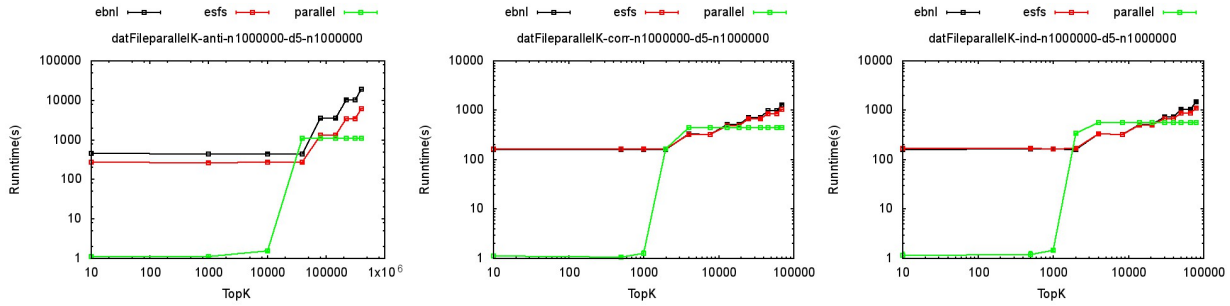
(a) Top-k: 1.000 (b) Top-k: 80.000

Fig. 6: Thread size - n: 1.000.000

Based on the overhead and the limit to the number of cores a higher thread size than the core numbers has always an increasing runtime. A lower thread size causes less parallel computation resulting in a higher runtime. Therefore, it can be concluded that neither a too low thread size nor a too high thread size results in the best runtimes. It is always recommended to use a thread size equal to the number of CPU cores.

6.3 Influence of the Top-k Value

Figure 7 shows that the parallel algorithm is very efficient for low k values compared to the sequential approaches. This can be explained by the fact that our algorithm runs in parallel and can be stopped after each block instead of each Skyline. After k exceeds the size of the first level, the whole multi-level Skyline needs to be computed. The parallel algorithm stays at the same runtime for higher k because the runtime can't increase since computing the whole multi-level Skyline is the only procession work to do. However, EBNL and ESFS need more iterations and comparisons for large ks. This leads to a better efficiency for higher k. Furthermore, it is shown that our parallel algorithm is more efficient for anti-correlated than correlated or independent data. This follows from the fact that anti-correlated data has more tuples which do not dominate each other and thus more comparisons need to be done in EBNL and ESFS.

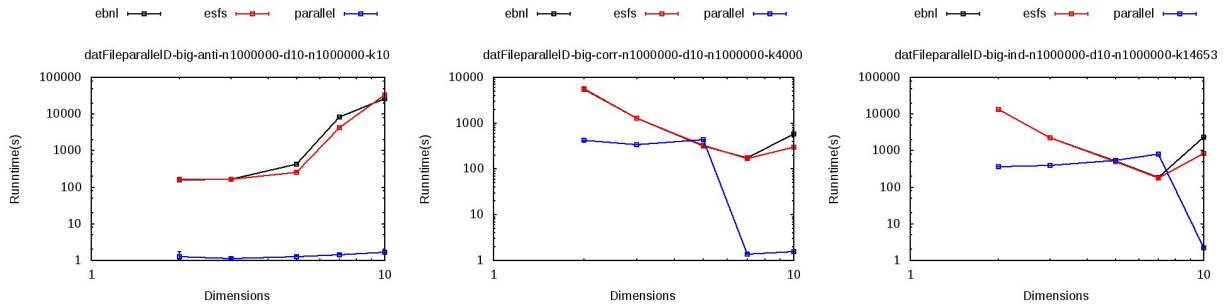


(a) anti (b) corr (c) ind

Fig. 7: Top-k - n: 1.000.000

6.4 Influence of the Number of Dimensions

As can be seen in Figure 8(a) the parallel algorithm is always more efficient than EBNL and ESFS for anti-correlated data and a small k. Increasing the number of dimensions causes the parallel algorithm to outperform the other algorithms. Similar insights can be found for correlated (see Figure 8(b)) and independent data (see Figure 8(c)).

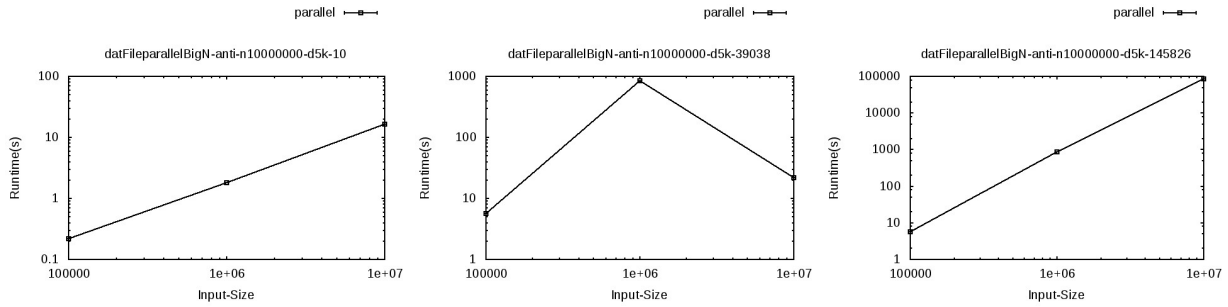


(a) anti - k: 10 (b) corr - k: 4000 (c) ind - k: 14653

Fig. 8: Dimension - n: 1.000.000

6.5 Influence of the Input Size

Since EBNL and ESFS run too long for an input size of 10.000.000, we solely tested the behavior of our parallel algorithm on increasing data. As shown in Figure 9(a) and 9(c), the runtime is linear to an increase of the input size. However, as seen in Figure 9(b) the runtime for 1.000.000 can be higher than the one for 10.000.000 if k is higher than the size of the first level of the multi-level Skyline for 1.000.000 tuples.



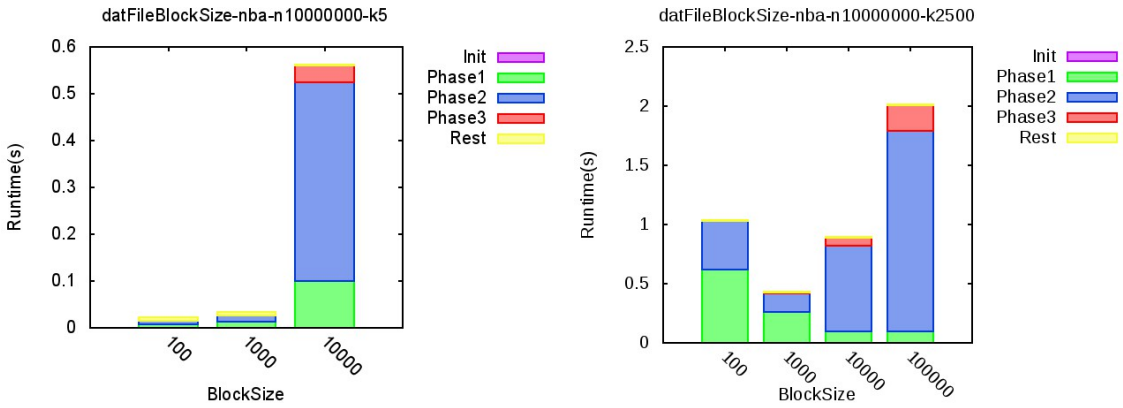
(a) Top-k: 10 (b) Top-k: 39038 (c) Top-k: 145826

Fig. 9: Different input size - anti - d: 5

6.6 Experiments on Real World Data

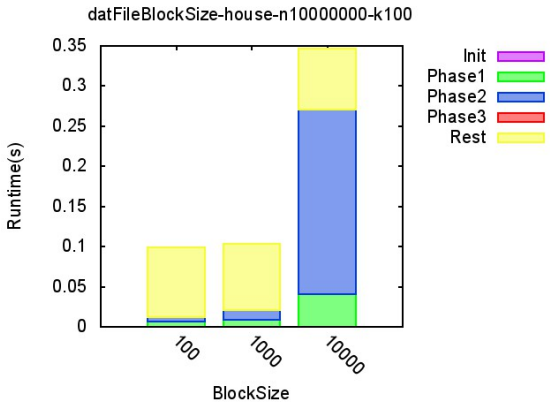
This section shows experiments on real data. For our benchmarks we used the well-known data sets NBA, House and Zillow. The data set NBA consists of 5 dimensions and 17265 entries. House is a data set with 6 dimensions and 127931 entries, and Zillow has 6 dimensions and 2245108 entries.

Influence of the Block Size For small data sets and small k, a block size of 100 or 1000 is the most efficient one as can bee see in Figures 10, 11, and 12(a). However, higher ks on larger data sets have an optimal block size of 10.000. This insight was found in Section 6.1. The experiment on the Zillow data set, cp. Figure 12(b), support this insight on real data.

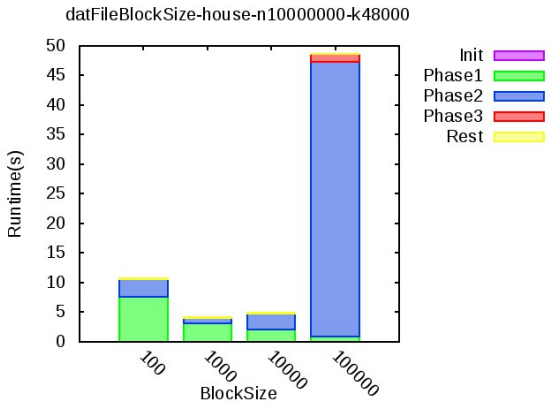


(a) Top-k: 5 (b) Top-k: 2.500

Fig. 10: Block size - NBA

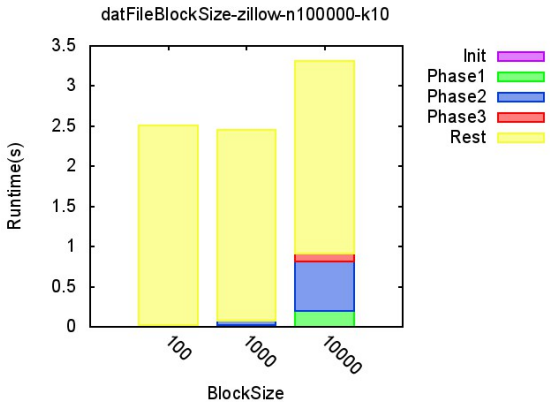


(a) Top-k: 100

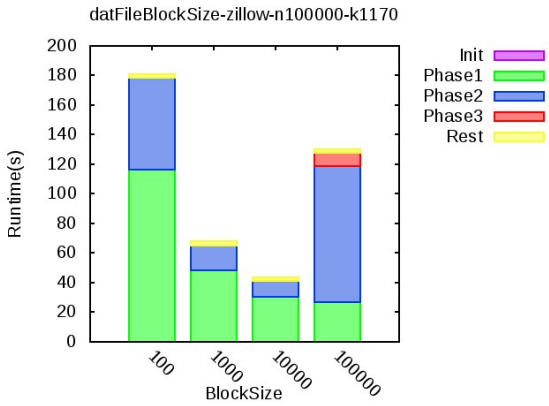


(b) Top-k: 48.000

Fig. 11: Block size - House



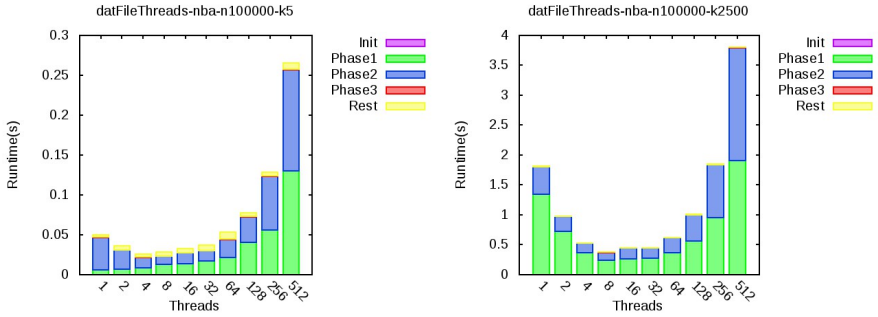
(a) Top-k: 10



(b) Top-k: 1170

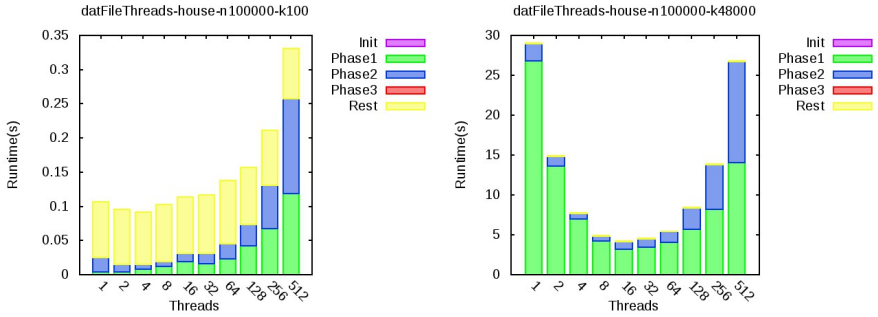
Fig. 12: Block size - Zillow

Influence of the Number of Threads In Section 6.2 we found out that the optimal thread size should be equal to the number of CPU cores. However, for small data sets this can be lower since there is less computation to do. Our experiments support these claims on real data as can be seen in Figures 13 - 15.



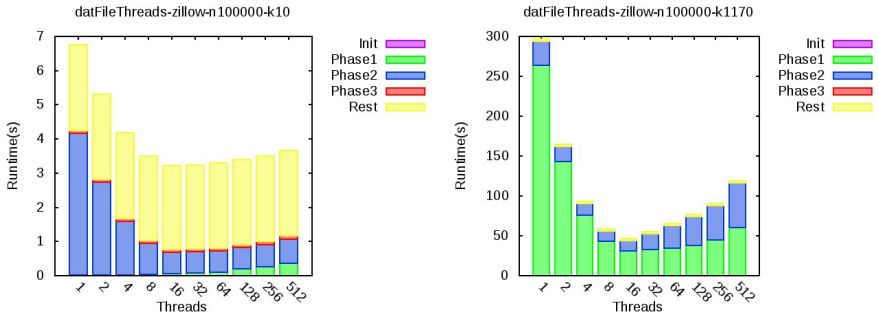
(a) Top-k: 5 (b) Top-k: 2500

Fig. 13: Threads - NBA



(a) k: 100 (b) k: 48000

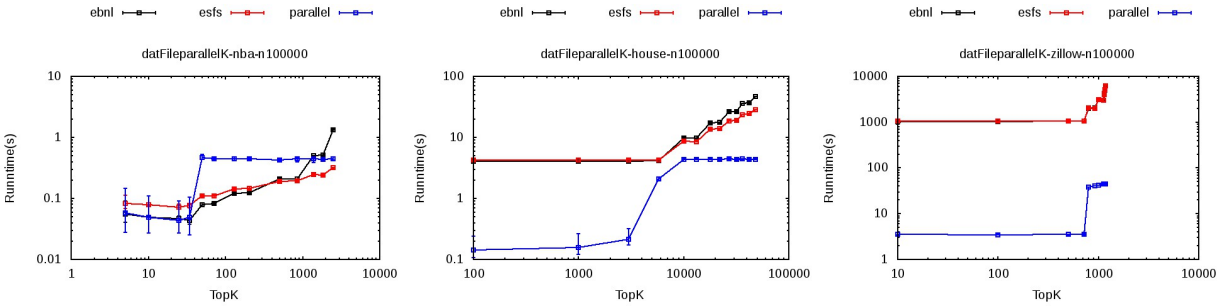
Fig. 14: Threads - House



(a) Top-k: 10 (b) Top-k: 1170

Fig. 15: Threads - Zillow

Influence of the Top-k Value Figure 16(a) shows that, for the NBA data set, the parallel algorithm performs better than the EBNL or ESFS for small k but worse for high k. For House (see Figure 16(b)) and Zillow (see Figure 16(c)) the parallel algorithm is even for k, which are higher than the size of the first level of the multi-level Skyline, more efficient than the other ones.



(a) NBA (b) House (c) Zillow

Fig. 16: Top-k on real data.

7 Summary and Outlook

In this technical report we presented a novel parallel algorithm for multi-level Skyline computation. Our algorithm rely on two parallel and one sequential phase. Our comprehensive approaches have shown that our parallel algorithm for multi-core architectures is more efficient for large data sets than existing approaches. For future work we will extend our algorithm such that it is able to compute multi-level Skylines in distributed environments as described in [MKEK15].

Acknowledgements

This work has been partially funded by the German Federal Ministry for Economic Affairs and Energy according to a decision by the German Bundestag, grant no. ZF4034402LF5.

We want to thank Monika Pichlmair for her support on implementing the algorithms and benchmark tests.

References

[BG04] W.-T. Balke and U. Güntzer. Multi-objective Query Processing for Database Systems. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 936–947. VLDB Endowment, 2004.

[BGG07] C. Brando, M. Goncalves, and V. González. Evaluating Top-k Skyline Queries over Relational Databases. In *DEXA '07: Proceedings of the 18th International Conference on Database and Expert Systems Applications*, volume 4653 of *Lecture Notes in Computer Science*, pages 254–263. Springer, 2007.

[BKS01] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proceedings of ICDE '01*, pages 421–430, Washington, DC, USA, 2001. IEEE.

[CCM13] J. Chomicki, P. Ciaccia, and N. Meneghetti. Skyline Queries, Front and Back. *SIGMOD*, 42(3):6–18, 2013.

[CGGL03] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In *Proceedings of ICDE '03*, pages 717–816, 2003.

[Cho03] J. Chomicki. Preference Formulas in Relational Queries. In *TODS '03: ACM Transactions on Database Systems*, volume 28, pages 427–466, New York, NY, USA, 2003. ACM Press.

[CSAB15] S. Chester, D. Sidlauskas, I. Assent, and K. S. Bøgh. Scalable Parallelization of Skyline Computation for Multi-Core Processors. In *Proceedings of ICDE '15*, pages 1083–1094, 2015.

[DPE08] S. Döring, T. Preisinger, and M. Endres. Advanced Preference Query Processing for E-Commerce. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 1457–1462, New York, NY, USA, 2008. ACM.

[EK11] M. Endres and W. Kießling. Skyline Snippets. In *FQAS '11: Proceedings of the 9th International Conference on Flexible Query Answering Systems*, Lecture Notes in Computer Science, pages 246–257, Berlin, Heidelberg, 2011. Springer-Verlag.

[EK14] M. Endres and W. Kießling. High Parallel Skyline Computation over Low-Cardinality Domains. In *Proceedings of ADBIS '14*, pages 97–111. Springer, 2014.

[EK15] M. Endres and W. Kießling. Parallel Skyline Computation Exploiting the Lattice Structure. *J. Database Manag.*, 26(4):18–43, 2015.

[End15] M. Endres. The Structure of Preference Orders. In *ADBIS '15: The 19th East European Conference in Advances in Databases and Information Systems*. Springer, 2015.

[EP15] M. Endres and T. Preisinger. Behind the Skyline. In *Proceedings of DBKDA '15*. IARIA, 2015.

[EP17] M. Endres and T. Preisinger. Beyond Skylines: Explicit Preferences. In *DASFAA '17: Database Systems for Advanced Applications - 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Proceedings, Part I*, pages 327–342, 2017.

[ERK14] M. Endres, P. Roocks, and W. Kießling. Algebraic Optimization of Grouped Preference Queries. In *Proceedings of IDEAS '14*, pages 247–256, New York, NY, USA, 2014. ACM.

[ERK15] M. Endres, P. Roocks, and W. Kießling. Scalagon: An Efficient Skyline Algorithm for all Seasons. In *Proceedings of DASFAA '15*, 2015.

[EW17] M. Endres and F. Weichmann. Index Structures for Preference Database Queries. In *FQAS '17: Proceedings of the 12th International Conference on Flexible Query Answering Systems*. Springer International Publishing, Cham, 2017.

[GSG07] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and Analyses for Maximal Vector Computation. *The VLDB Journal*, 16(1):5–28, 2007.

[GV05a] M. Goncalves and M.-E. Vidal. Preferred Skyline: A Hybrid Approach Between SQLf and Skyline. In *DEXA '05: 16th International Conference on Database and Expert Systems Applications*, volume 3588 of *Lecture Notes in Computer Science*, pages 375–384. Springer, 2005.

[GV05b] M. Goncalves and M.-E. Vidal. Top-k Skyline: A Unified Approach. In *OTM Workshops*, pages 790–799, 2005.

[GV09] M. Goncalves and M.-E. Vidal. Reaching the Top of the Skyline: An Efficient Indexed Algorithm for Top-k Skyline Queries. In *Database and Expert Systems Applications*, volume 5690 of *Lecture Notes in Computer Science*, pages 471–485. Springer Berlin Heidelberg, 2009.

[KEK17] J. Kastner, M. Endres, and W. Kießling. A Pareto-Dominant Clustering Approach for Pareto-Frontiers. In *DOLAP '17: Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, March 21-24, 2017.*, 2017.

[KEW11] W. Kießling, M. Endres, and F. Wenzel. The Preference SQL System - An Overview. *Bulletin of the Technical Commitee on Data Engineering, IEEE Computer Society*, 34(2):11–18, 2011.

[Kie02] W. Kießling. Foundations of Preferences in Database Systems. In *Proceedings of VLDB '02*, pages 311–322, Hong Kong, China, 2002. VLDB.

[LVDN] S. Liknes, A. Vlachou, C. Doulkeridis, and K. Nørvåg. APSkyline: Improved Skyline Computation for Multicore Architectures. In *Proc. of DASFAA '14*.

[LYH09] J. Lee, G-W. You, and S-W. Hwang. Personalized Top-k Skyline Queries in High-Dimensional Space. *Information Systems*, 34(1):45–61, March 2009.

[LYLC06] E. Lo, K. Y. Yip, K.-I. Lin, and D. W. Cheung. Progressive Skylining over Web-Accessible Databases. *IEEE TKDE*, 57(2):122–147, 2006.

[LZLL07] K. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the Skyline in Z Order. In *Proceedings of VLDB '07*, pages 279–290. VLDB Endowment, 2007.

[MKEK15] S. Mandl, O. Kozachuk, M. Endres, and W. Kießling. Preference Analytics in EXASolution. In *Proceedings of BTW '15*, 2015.

[MPJ07] M. Morse, J. M. Patel, and H. V. Jagadish. Efficient Skyline Computation over Low-Cardinality Domains. In *Proceedings of VLDB '07*, pages 267–278, 2007.

[PE15] T. Preisinger and M. Endres. Looking for the Best, but not too Many of Them: Multi-Level and Top-k Skylines. *International Journal on Advances in Software*, 8(3&4):467–480, 2015.

[PK07] T. Preisinger and W. Kießling. The Hexagon Algorithm for Evaluating Pareto Preference Queries. In *Proceedings of MPref '07*, 2007.

[PP09] Q. Li ZY Chen J. Bian P. Pan, YQ Sun. The Top-k Skyline Query in Pervasive Computing Environments. In *Pervasive Computing (JCPC), 2009 Joint Conferences on*, pages 335–338. IEEE, 2009.

[PTFS03] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *Proceedings of SIGMOD '03*, pages 467–478. ACM, 2003.

[RJVDN10] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Efficient Processing of Top-k Spatial Preference Queries. In *Proceedings of the VLDB Endowment*. VLDB Endowment, November 2010.

[SCL10] I-F. Su, Y.-C. Chung, and C. Lee. Top-k Combinatorial Skyline Queries. In *Database Systems for Advanced Applications*, pages 79–93. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2010.

[SLB10] J. Selke, C. Lofi, and W.-T. Balke. Highly Scalable Multiprocessing Algorithms for Preference-Based Database Retrieval. In *Proceedings of DASFAA '10*, volume 5982 of *LNCS*, pages 246–260. Springer, 2010.

[TXP07] Y. Tao, X. Xiao, and J. Pei. Efficient Skyline and Top-k Retrieval in Subspaces. *IEEE TKDE*, 19(8):1072–1088, 2007.

[VDNV08] A. Vlachou, C. Doulkeridis, K. Nørvåg, and M. Vazirgiannis. Skyline-based Peer-to-Peer Top-k Query Processing. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1421–1423, Washington, DC, USA, 2008. IEEE Computer Society.

[YAY12] A. Yu, P. K. Agarwal, and J. Yang. Processing a Large Number of Continuous Preference Top-k Queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 397–408. ACM, 2012.

[YDMV07] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis. Top-k Spatial Preference Queries. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1076–1085. IEEE, 2007.

[ZZL+11] Y. Zhang, W. Zhang, X. Lin, B. Jiang, and J. Pei. Ranking Uncertain Sky: The Probabilistic Top-k Skyline Operator. *Information Systems*, 36(5):898–915, July 2011.