Looking for the Best, but not too Many of Them: Multi-Level and Top-k Skylines

Timotheus Preisinger

DEVnet Holding GmbH
Grünwald, Germany
Email: t.preisinger@devnet.de

Markus Endres

University of Augsburg
Augsburg, Germany
Email: markus.endres@acm.org

***Abstract*—The problem of *Skyline* computation has attracted considerable research attention in the last decade. A *Skyline* query selects those tuples from a dataset that are optimal with respect to a set of designated preference attributes. However, the number of Skyline answers may be smaller than required by the user, who needs at least k. Given a dataset, a *top-k Skyline* query returns the *k most interesting* elements of the Skyline query based on some kind of user-defined preference. That said, in some cases, not only the Pareto frontier is of interest, but also the stratum 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 present the concept of *multi-level Skylines* for the computation of different strata and we discuss *top-k* Skyline queries in detail. Our algorithms rely on the lattice structure constructed by a Skyline query over low-cardinality domains. In addition we present external versions of our algorithms such that it is not necessary to store the complete lattice in main memory. We demonstrate through extensive experimentation on synthetic and real datasets that our algorithms can result in a significant performance advantage over existing techniques.**

Keywords–Skyline; Preferences; Multi-level; Top-k; Lattice.

I. INTRODUCTION

Information systems of different types use various techniques to rank query answers. In such systems users are often interested in the most important (top-k) and most preferred query answers in the potentially huge answer space. Different preference-based query languages have been defined to support the bases for discriminating poor quality data and to express user's preference criteria on top-k [1][2][3].

The Skyline operator for example [4] has emerged as an important and popular technique for searching the best objects in multi-dimensional datasets. A Skyline query selects those objects from a dataset D that are not dominated by any others. An object p having d attributes (dimensions) dominates an object q, if p is strictly better than q in at least one dimension and not worse than q in all other dimensions, for a defined comparison function. Without loss of generality, we consider subsets of $\mathbb{R}^d$ in which we search for Skylines with respect to (abbr. w.r.t.) the natural order $\leq$ in each dimension.

Example 1. *The most cited example on Skyline queries is the 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. Table I presents a sample dataset with its visualization in Figure 1. Each hotel is represented as a point in the two-dimensional space of price and distance to the beach. Interesting are all hotels that are not worse than any other hotel in these both dimensions.*

TABLE I. Sample dataset 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 $\mathcal{S}$. From the Skyline, one can now make the final decision, thereby weighing the personal preferences for price and distance.

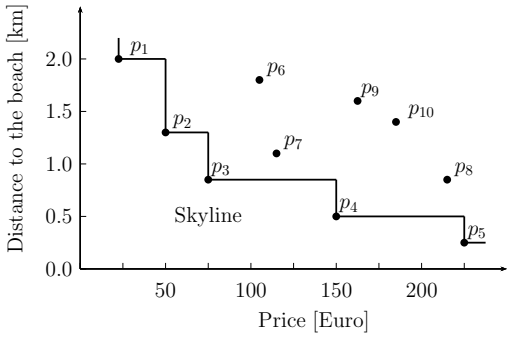


Figure 1. Skyline example.

Most of the work on Skyline computation has focused on the development of efficient algorithms for preference evaluation ([3] gives an overview). The most prominent algorithms are characterized by a tuple-to-tuple comparison-based approach [4][5]. Based on this, several algorithms have been published in the last decade, e.g., NN (Nearest Neighbor) [6], BBS (Branch and Bound Skyline) [7], SFS (Sort-Filter Skyline) [8], or LESS (Linear Elimination-Sort for Skyline) [9], just to name a few.

Unfortunately, the size of the Skyline $\mathcal{S}$ can be very small (e.g., in low-dimensional spaces). Hence, a user might want to see the *next best objects behind the Skyline*.

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

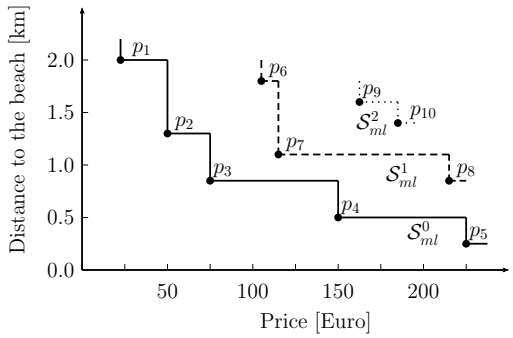


Figure 2. Multi-level Skylines.

Furthermore, in the presence of high-dimensional Skyline spaces, the size of the Skyline $\mathcal{S}$ can still be very large, making it unfeasible for users to process this set of objects [3]. Hence, a user might want to see the *top-k* objects. That means a maximum of k objects out of the complete Skyline set if $|\mathcal{S}| >= k$, or, for $|\mathcal{S}| < k$, use the Skyline set plus the next best objects such that there will be k results. In the previous example a *top-3* Skyline query would identify, e.g., p_1, p_2, and p_3, whereas in a *top-10* query it is necessary to consider the second and third stratum to identify p_6, p_7, p_8, p_9, and p_{10} as additional Skyline points.

In this work, we propose evaluation strategies for *multi-level and top-k Skyline* queries, which do not depend on tuple comparisons. For this we generalize the well-known Skyline queries to *multi-level Skylines* $\mathcal{S}_{ml}$. We present an efficient algorithm to compute the *l-th stratum* of a Skyline query exploiting the lattice structure constructed over low-cardinality domains. Following [3][10][11], many Skyline applications involve domains with small cardinalities – these cardinalities are either inherently small (such as star ratings for hotels), or can naturally be mapped to low-cardinality domains (such as price ranges on hotels). In addition, we propose an evaluation strategy for *top-k Skyline* queries, which is based on the multi-level approach.

This paper is an extended version of [1] and additionally contains a deeper background on lattice-based Skyline computation, additional theoretical results, detailed description of the multi-level and top-k Skyline algorithms, examples, more comprehensive experiments, and extended related work. In addition we provide a section about an external implementation of our algorithms such that they do not rely on large main memory.

The remainder of this paper is organized as follows: In Section II we present the formal background. Based on this background we will discuss *multi-level Skyline* computation in Section III and *top-k Skyline* computation in Section IV. In Section V we present an external version of our algorithm. Section VI contains some remarks. We conduct an extensive performance evaluation on synthetic and real datasets in Section VII. Section VIII contains related work, and Section IX concludes our paper.

II. Skyline Queries Revisited

In this section, we revisit the problem of Skyline computation and shortly describe the Lattice Skyline approach, since this is the basis of our algorithms.

A. Skyline Queries

The aim of a Skyline query is to find the *best objects* in a dataset D, i.e., $\mathcal{S}(D)$. More formally:

Definition 1 (Dominance and Indifference)**.** *Assume a set of vectors $D \subseteq \mathbb{R}^d$. Given $x = (x_1, ..., x_d)$, $y = (y_1, ..., y_d) \in D$, x* dominates *y on D, denoted as $x <_\otimes y$, if the following holds:*

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

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

Note that following Definition 1 we consider subsets of $\mathbb{R}^d$ in that we search for the Skyline w.r.t. the natural order $\leq$ in each dimension. Equation (1) is also known as *Pareto ordering* [12][13][14][15].

Definition 2 (Skyline $\mathcal{S}$)**.** *The* Skyline $\mathcal{S}(D)$ *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 we prefer the minimal values in each domain and write $x <_\otimes y$ if x is better than y.

Note that an extension to arbitrary orders specified by utility functions is obvious and that Skylines are not restricted to numerical domains [16]. For any universe Ω and orderings $<_i \in (\Omega \times \Omega)$ $(i \in \{1, ..., d\})$ the Skyline w.r.t. $<_i$ can be computed, if there exist scoring functions $g_i : \Omega \to \mathbb{R}$ for all $i \in \{1, ..., d\}$ such that $x <_i y \Leftrightarrow g_i(x) < g_i(y)$. Then the Skyline of a set $M \subseteq \Omega$ w.r.t. $(<_i)_{i=1,...,d}$ is equivalent to the Skyline of $\{(g(x_1), ..., g(x_d)) \mid x \in M\}$.

In general, algorithms of the block-nested-loop class (BNL) [4] are probably the best known algorithms for computing Skylines. They are characterized by a tuple-to-tuple comparison-based approach, hence having a worst case complexity of $\mathcal{O}(n^2)$, and a best case complexity of the order $\mathcal{O}(n)$; n being the number of input tuples, cf. [9]. The major advantage of a BNL-style algorithm is its simplicity and suitability for computing the maxima of arbitrary partial orders. Furthermore, a multitude of optimization techniques [8][9] and parallel variants [17][18][19][20] have been developed in the last decade.

B. Lattice Skyline Revisited

Lattice-based algorithms depend on the lattice structure constructed by a Skyline query over low-cardinality domains. An attribute domain $\mathrm{dom}(S)$ is said to be *low-cardinality* if its value is drawn from a set $S = \{s_1, \ldots s_m\}$, such that the set cardinality m is small. Examples for such algorithms are

Lattice Skyline [11] and *Hexagon* [10], both having a worst case linear time complexity. Both algorithms follow the same idea: the partial order imposed by a Skyline query constitutes a *lattice*.

Definition 3 (Lattice [21])**.** *A partially ordered set D with operator '$<_{\otimes}$' is a* lattice *if $\forall x, y \in D$, the set $\{x, y\}$ has a* least upper bound *and a* greatest lower bound *in D. If a least upper bound and a greatest lower bound is defined for all subsets of D, we have a complete lattice.*

The proof that a Skyline query constitutes a lattice can be found in [21].

Visualization of such lattices is often done using *Better-Than-Graphs* (*BTG*) [22], graphs in which edges state dominance. The nodes in the BTG represent *equivalence classes*. Each equivalence class contains the objects mapped to the same feature vector of the Skyline query. All values in the same equivalence class are indifferent and considered substitutable.

Before we consider an example, we need the notion of $\max(A)$.

Definition 4 ($\max(A)$)**.** $\max(A)$ *is the* maximum value *for a preference on the attribute A.*

Example 3. *An example of a BTG over a 2-dimensional space is shown in Figure 3 where $[0..2] \times [0..4]$ describes a domain of integers where attribute $A_1 \in \{0, 1, 2\}$ and $A_2 \in \{0, 1, 2, 3, 4\}$ (abbr. $[2; 4]$, $\max(A_1) = 2$, $\max(A_2) = 4$). The arrows show the dominance relationship between elements of the lattice. The node $(0, 0)$ presents the* best node*, i.e., the least upper bound, whereas $(2, 4)$ is the* worst node*. The bold numbers next to each node are* unique identifiers *(ID) for each node in the lattice, cp. [10]. Nodes having the same level in the BTG are indifferent, i.e., for example, that neither the objects in the node $(0, 4)$ are better than the objects in $(1, 3)$ nor vice versa. A dataset D does not necessarily contain tuples for each lattice node. In Figure 3, the gray nodes are occupied (*non-empty*) with elements from the dataset whereas the white nodes have no element (*empty*).*

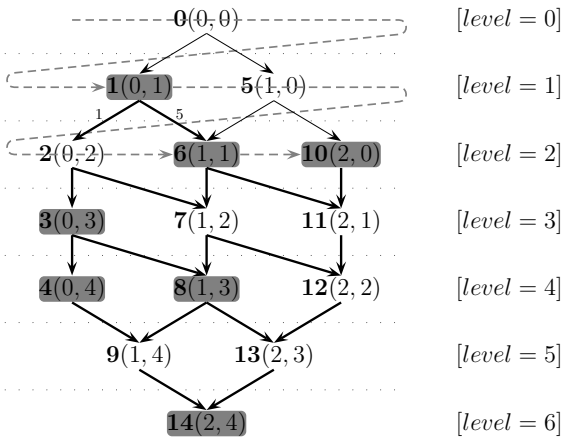


Figure 3. Lattice over $[0..2] \times [0..4]$.

The method to obtain the Skyline can be visualized using the BTG. The elements of the dataset D that compose the Skyline are those in the BTG that have no path leading to them from another non-empty node in D*. In Figure 3, these are the nodes $(0, 1)$ and $(2, 0)$. All other nodes have direct or transitive edges from these both nodes, and therefore are* dominated*.*

When consing lattice algorithms the question arises how to map tuples t from a dataset D to the lattice structure. In [11] the authors use a function $F(t)$, which denotes a one-to-one mapping of an element $t \in D$ to a position in the BTG. For example, in the boolean case one can use the binary value of the boolean attributes to determine the array position, i.e., if $d = 3$, then the element *(true, false, true)* $\in D$ is represented by position 5 (101 in binary representation) in the BTG. A more general approach is presented in [10], where the position of a tuple is computed as below and also serves as the *unique identifier* (ID) mentioned in Example 3.

Lemma 1 (Edge Weights and Unique Node IDs)**.** *Let $\mathcal{S}(D)$ be a Skyline query over a* d*-dimensional low-cardinality domain $\mathrm{dom}(A) := \mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_d)$, and $a = (a_1, \ldots, a_d) \in \mathrm{dom}(A)$.*

a) The weight *of an edge in the BTG expressing dominance between two direct connected nodes w.r.t. any attribute A_i is characterized by*

$$\mathrm{weight}(A_i) := \prod_{j=i+1}^{d} (\max(A_j) + 1) \tag{3}$$

For $j > d$ we set $\mathrm{weight}(A_i) = 1$.

b) The unique identifier *(ID) for $a \in \mathrm{dom}(A)$ is given by*

$$\mathrm{ID}(a) = \sum_{i=1}^{d} (\mathrm{weight}(A_i) \cdot a_i) \tag{4}$$

The proof of Lemma 1 can be found in [10].

Example 4. *Reconsider Example 3. The edge weights of the node $a := (a_1, a_2) = (0, 1)$ are* $\mathrm{weight}(a_1) = 4 + 1 = 5$ *and* $\mathrm{weight}(a_2) = 1$ *as annotated in Figure 3. Hence, the unique identifier is* $\mathrm{ID}(a) = 5 \cdot 0 + 1 \cdot 1 = \mathbf{1}$*, the bold number left of the node.*

Note that the edge weights are also used to find the *direct dominated* nodes of a given node a. Just add the different edge weights to the unique identifier and one will get all direct dominated nodes, e.g., 1+1 = **2** and 1 + 5 = **6**, both are dominated by node **1** in Figure 3. The pseudocode to find all direct dominated nodes is depicted in Algorithm 1 and is straightforward.

Lattice based algorithms exploit these observations to find the Skyline of a dataset over the space of vectors drawn from low-cardinality domains and in general consist of three phases. The pseudocode can be found in Algorithm 2. For details we refer to [10] and [11].

1) **Phase 1:** The *Construction Phase* initializes the data structures. The lattice is represented by an *array* in main memory (line 3 in Algorithm 2). Each position in the array stands for one node ID in the lattice. Initially, all nodes of the lattice are marked as *empty* and *not dominated*.
2) **Phase 2:** In the *Adding Phase* the algorithm determines for each element $t \in D$ the unique ID and therefore the node of the lattice that corresponds to t. This node will be marked as *non-empty* (line 7).

Algorithm 1 getDirectDominatedNodesBy(a)

Input: Node a.
Output: List of immediate dominated nodes.

```
1: function GETDIRECTDOMINATEDNODESBY(a)
2:     nodes ← list() // empty list to store dominated nodes
3:     // loop over the A_i
4:     for i ← 1, ..., d do
5:         // check if there is an edge for A_i
6:         domNode ← ID(a) + weight(A_i)
7:         if domNode is valid, then
8:             // domNode is in the next level and inside the BTG
9:             nodes.add(domNode)
10:        end if
11:    end for
12:    return nodes
13: end function
```

3) **Phase 3:** After all tuples have been processed, in the *Removal Phase* dominated nodes are identified. The nodes of the lattice that are marked as *non-empty* and which are not reachable by the transitive dominance relationship from any other *non-empty* node represent the Skyline values. Nodes that are *non-empty* but are reachable by the dominance relationship are marked *dominated* to distinguish them from present Skyline values.
From an algorithmic point of view this is done by a combination of *breadth-first traversal* (BFT) and *depth-first traversal* (DFT). The nodes of the lattice are visited level-by-level in a BFT (the dashed line in Figure 3, line 10 in Algorithm 2). Each time a *non-empty* and *not dominated* node is found, a DFT will start to mark dominated nodes as *dominated* (lines 12 – 17). The DFT does not need to explore branches already marked as *dominated*. The BFT can stop after processing a whole level not containing *empty* nodes hence marking the end of Phase 3.
For example, the node $(0, 1)$ in Figure 3 is not empty. The DFT recursively walks down and marks all dominated nodes as *dominated* (thick black arrows). After the BFT has finished, the *non-empty* and *not dominated* nodes (here $(0, 1)$ and $(2, 0)$) contain the Skyline objects.

III. Multi-Level Skyline Computation

In some cases it is necessary to return not only the best tuples as in common Skyline computation, but also to retrieve tuples directly dominated by those of the Skyline set (the second *stratum*), i.e., the tuples *behind the Skyline*. Following this method transitively, the input is partitioned into multiple levels (*strata*) in a way resembling the elements' quality w.r.t. the search preferences. In this section, we introduce the concept of *multi-level Skylines* and present an algorithm for efficient computation of iterated preferences in linear time.

A. Background

We extend Definition 2 of the Skyline by a level value to form *multi-level Skyline* ($\mathcal{S}_{ml}$) sets.

Definition 5 (Multi-Level Skyline $\mathcal{S}_{ml}$)**.** *The multi-level Skyline set of level l (i.e., the l-th stratum) for a dataset D is defined as*

$$\begin{aligned} \mathcal{S}^0_{ml}(D) &:= \mathcal{S}(D) \\ \mathcal{S}^l_{ml} &:= \mathcal{S}\left(D \setminus \bigcup_{i=0}^{l-1} \mathcal{S}^i_{ml}(D)\right) \end{aligned}$$

Algorithm 2 Lattice Skyline (cp. [10][11])

Input: Dataset D with n tuples over d low-cardinality attributes, Array BTG of size V, V_i is the cardinality of dimension i.
Output: Skyline points.

```
1: // Phase 1: Construction Phase
2: Let V be the number of entries in the lattice, V ← V_1, ·, ... · V_d
3: Let BTG be an array of size V holding the different designators
   empty, non-empty, dominated, initialized to empty
4: // Let ID(t) be the unique identifier of a tuple t ∈ D and the
   index position in BTG.
5: // Phase 2: Adding Phase
6: for all t ∈ D do
7:     Set BTG[ID(t)] to non-empty
8: end for
9: // Phase 3: Removal Phase
10: for i ← 0 ... V in a BFT do
11:     // Find all direct dominated nodes
12:     for all g ∈ getDirectDominatedNodesBy(BTG[i]) do
13:         if BTG[i] == (non-empty or dominated) then
14:             BTG[g] ← dominated
15:             do a DFT to mark successors as dominated
16:         end if
17:     end for
18: end for
19: // Output Skyline
20: for all t ∈ D do
21:     if BTG[ID(t)] == non-empty then
22:         output t as a Skyline point
23:     end if
24: end for
```

Thereby $\mathcal{S}^0_{ml}(D)$ is identical to the standard Skyline $\mathcal{S}(D)$ from Definition 2, and $\mathcal{S}^{l_{max}}_{ml}$ denotes the non-empty set with the highest level.

Example 5. *For example, the query $\mathcal{S}^1_{ml}(D)$ on our hotel sample dataset computes the set of "second-best" tuples in the dataset D, i.e., the second stratum consisting of the objects p_6, p_7, and p_8 as depicted in Figure 2, dashed line. The query $\mathcal{S}^2_{ml}(D)$ returns p_9, p_{10}.*

Therefore, by iterating the Skyline operator one can rank the tuples in a given relation instance. Before we present our algorithm for multi-level Skyline computation we prove some properties.

Lemma 2. *For each tuple t in a finite dataset D, there is exactly one $\mathcal{S}^l_{ml}$ set it belongs to:*

$$\forall t \in D : \left(\exists! l : t \in \mathcal{S}^l_{ml}(D)\right) \tag{5}$$

Proof: A Skyline query on $D \neq \emptyset$ never yields an empty result, i.e., $\mathcal{S}(D) \neq \emptyset$. Starting at 0, for each $l = 0, 1, 2, \ldots$ the input dataset diminishes as all selection results for smaller values of l are removed from the input. Since $D \setminus \bigcup_{i=0}^{l} \mathcal{S}^i_{ml}(D) \subset D \setminus \bigcup_{i=0}^{l-1} \mathcal{S}^i_{ml}(D)$ and $|D|$ is finite, there has to be some l_{max} for which the following holds:

$$\bigcup_{i=0}^{l_{max}-1} \mathcal{S}^i_{ml}(D) \subset D \ \wedge \ \bigcup_{i=0}^{l_{max}} \mathcal{S}^i_{ml}(D) = D$$

So each tuple in D belongs to exactly one $\mathcal{S}^i_{ml}(D)$. ■

Lemma 2 shows that all tuples in a dataset belong to a $\mathcal{S}_{ml}$ set of some level. So a kind of order on D w.r.t. the Skyline query is induced.

Lemma 3. *All elements of $\mathcal{S}^l_{ml}(D)$ are dominated by elements of $\mathcal{S}^i_{ml}(D)$ for all $i < l$:*

$$\forall y \in \mathcal{S}^l_{ml}(D) : (\exists x \in \mathcal{S}^i_{ml}(D) : x <_{\otimes} y)) \quad \textit{if} \quad i < l \qquad (6)$$

Proof: Consider a tuple $y \in \mathcal{S}^l_{ml}(D)$ that is not dominated by any element of $\mathcal{S}^i_{ml}(D)$ for $i < l$. Following Definition 5, $y \in \mathcal{S}^i_{ml}(D)$. This is a contradiction. ■

For every Skyline query on $D \neq \emptyset$ there is at least a $\mathcal{S}_{ml}$ set of level 0. If it is the only one, no tuple in D is worse than any other w.r.t. the preference. Just as well, it is possible that all tuples in D belong to $\mathcal{S}_{ml}$ sets of different levels. The Skyline query then defines a total order on the elements of D.

For each node x in the BTG, we can determine the stratum l of the $\mathcal{S}^l_{ml}$ set it belongs to. Of course, all tuples in one equivalence class (which is represented by one node) are elements of the same $\mathcal{S}^l_{ml}$ set. To find the $\mathcal{S}^l_{ml}$ set for each node, we start at level 0 at the top node of the better-than graph. All tuples belonging to the standard Skyline set have a $\mathcal{S}^0_{ml}$ set level of 0. As the following lemma will show, the level of each tuple is the highest level of all tuples dominating it, increased by one:

Lemma 4 ($\mathcal{S}^l_{ml}$ set level for an object)**.** *For an object $t \in D$ (or the BTG node representing its level values), $l(t) : \mathrm{dom}(A) \to \mathbb{N}_0$ can be computed as follows:*

$$l(t) := \begin{cases} 0 & \iff t \in \mathcal{S}^0_{ml}(D) \\ 1 + \max(\{l(s) | s \in D \wedge t <_{\otimes} s\}) & \\ & \iff t \notin \mathcal{S}^0_{ml}(D) \end{cases}$$

Proof: This follows from Definition 5 and Lemma 3. ■

With these concepts, we are now able to adjust lattice-based Skyline algorithms to compute multi-level Skyline sets.

B. The Multi-Level Lattice Skyline Algorithm (MLLS)

We will now see how the lattice based Skyline algorithms described in Section II-B can be adjusted to support multi-level Skyline computation. We call this algorithm *Multi-Level Lattice Skyline* (MLLS). The first two phases of the standard lattice algorithms, *construction* and *adding*, remain unchanged. Modifications have to be done solely in the *removal phase*. Actually, as dominated nodes are not "removed" anymore, the *removal phase* is replaced by a *node classification phase*, cp. Algorithm 3.

The *classification phase* uses the same breadth-first and depth-first traversal as the original lattice Skyline algorithms. We need the node states *empty* and *non-empty*. In addition, we need to store a temporary value tmp_{ml} for the level of the $\mathcal{S}_{ml}$ set a node belongs to currently. When a node n is reached, we reset the tmp_{ml} values for the nodes $v_1, v_2, \ldots$, that are directly dominated by n. The value $\mathrm{tmp}_{ml}(v_i)$ for a node v_i is computed as follows:

$$\mathrm{tmp}_{ml}(v_i) = \begin{cases} \max(\mathrm{tmp}_{ml}(v_i),\ \mathrm{tmp}_{ml}(n)) & \Leftrightarrow \ n \text{ is empty} \\ \max(\mathrm{tmp}_{ml}(v_i),\ \mathrm{tmp}_{ml}(n)+1) & \Leftrightarrow \ n \text{ is not empty} \end{cases}$$

Algorithm 3 Multi-Level Skyline – Classification Phase

Input: Better-Than Graph (BTG)
Output: list of $\mathcal{S}_{ml}$ sets

```
1:  function CLASSIFY(BTG)
2:      S_ml ← list<list>() // initialize list to store S_ml sets
3:      tmp_ml[|BTG|] ← 0 // initialize tmp_ml array with 0's
4:      // iterate over all nodes n (BFT), start with node ID 0
5:      n ← node(ID = 0)
6:      repeat
7:          // use offset for tmp_ml computation
8:          offset ← n.isEmpty() ? 0 : 1
9:          // let domNodes be the list of direct dominated nodes
10:         domNodes ← getDirectDominatedNodesBy(n)
11:         for all v in domNodes do // compute tmp_ml
12:             tmp_ml(v) ← max(tmp_ml(v), tmp_ml(n) + offset)
13:         end for
14:         // node not empty, add objects to S_ml sets
15:         if !n.isEmpty() then
16:             i ← tmp_ml[n]
17:             // add all elements in node n to the S^i_ml set
18:             S_ml.addAll(i, n.getElements())
19:         end if
20:         n ← nextNode() // next node in BFT
21:     until n == NIL // repeat until end of BTG is reached
22:     return S_ml
23: end function
```

In Algorithm 3, for a more convenient and efficient access to each of the $\mathcal{S}_{ml}$ sets after the classification phase, we generate a list of nodes belonging to each $\mathcal{S}_{ml}$ set while walking through the BTG. For this, we initialize a list of lists, which will store the $\mathcal{S}^i_{ml}$ sets for each level i and an *array* of size of the BTG for the tmp_{ml} levels values (lines 2–3). Then we start the BFT at node 0 (line 5). If the current node n is empty we set an offset to 0, otherwise to 1 (line 8). The function `getDirectDominatedNodesBy()` retrieves all nodes directly dominated by n (cp. Algorithm 1). The complexity of this function is given by the number of Skyline dimensions as for each of them not more than one node can be dominated and we only visit directly dominated nodes (so the DFT ends at depth 1). The actual complexity of finding each of the directly dominated nodes or a node's successor in the BFT is specific to the representation of the BTG in memory, but can be assumed as $\mathcal{O}(1)$ [10][11]. For all direct dominated nodes compute the tmp_{ml} value in (lines 11–13). Afterward, if the node n contains elements from the input dataset, we retrieve for $\mathcal{S}^i_{ml}$ the level i the elements belongs to (line 16) and add all elements of the node n to the $\mathcal{S}^i_{ml}$ building up the multi-level Skyline sets (line 18). We continue with the next node in the BFT (line 20) until the end of the BTG is reached (line 21). The result is a list of $\mathcal{S}^i_{ml}$ sets.

Example 6. *Figure 4 visualizes an example of Algorithm 3.*

Since node 0 is empty, the first relevant node is 1. Therefore we set $\mathrm{tmp}_{ml}[1] = 0$ and add 1 to all direct dominated nodes, i.e., $\mathrm{tmp}_{ml}[2] = \mathrm{tmp}_{ml}[6] = 1$. We continue with node 5, which does not affect anything (the offset for the node is 0 and hence the tmp_{ml} values for the dominated nodes 6 and 10 remain unchanged). Since node 2 is empty, we set $\mathrm{tmp}_{ml}[3] = \mathrm{tmp}_{ml}[7] = 1$. Node 6 has already $\mathrm{tmp}_{ml}[6] = 1$. The next node is 10, which still has $\mathrm{tmp}_{ml}[10] = 0$. Node 3 sets $\mathrm{tmp}_{ml}[4] = \mathrm{tmp}_{ml}[8] = 2$, and so on. After the BFT has finished we have 4 $\mathcal{S}_{ml}$ sets.

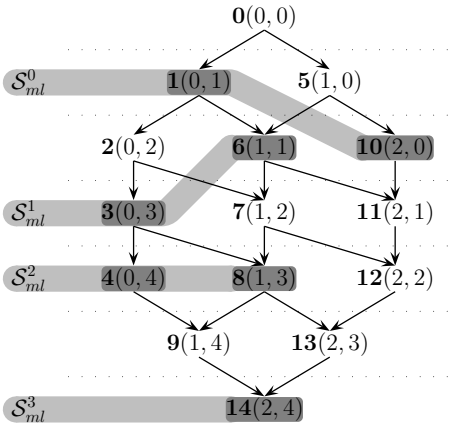


Figure 4. Multi-level Skylines $\mathcal{S}^l_{ml}$.

Note that our multi-level Skyline algorithm has the same runtime complexity as the original Lattice Skyline algorithms [10][11]. In MLLS the construction and adding phase are unchanged in comparison to Lattice Skyline. The classification phase is a simple modification of the original removal phase without any additional overhead. Therefore, we state a linear runtime complexity of $\mathcal{O}(dV + dn)$, where d is the dimensionality, n is the number of input tuples, and V is the product of the cardinalities of the low-cardinality domains from which the attributes are drawn.

IV. Top-k Skyline Computation

The concept of *top-k* ranking is used to rank tuples according to some score function and to return a maximum of *k* objects [23]. On the other hand, Skyline retrieves tuples where all criteria are equally important concerning some user preference [4]. However, the number of Skyline answers may be smaller than required by the user, for whom k are needed. Therefore, *top-k Skyline* was defined as a unified language to integrate them [24][25].

A. Background

Top-k Skyline allows to get exactly the top k from a partially ordered set stratified into subsets of non-dominated tuples. The idea is to partition the set into subsets (strata, multi-level Skyline sets) consisting of non-dominated tuples and to produce the top-k of these partitions.

In general, existing solutions calculate the first stratum with some sort of post-processing [24][25][26]. That means, after identifying the first stratum $\mathcal{S}^0_{ml}(D)$, they remove the contained objects from the original input dataset D and continue Skyline computation on the reduced data. Hence, the second stratum is $\mathcal{S}^1_{ml} = \mathcal{S}(D \setminus \mathcal{S}(D))$. This workflow is continued until k objects are found. Definition 6 outlines the three different cases which might occur:

Definition 6 (Top-k Skyline)**.** *A* top-k Skyline *query* $\mathcal{S}^k_{tk}(D)$ *on an input dataset* 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* $\mathcal{S}^k_{tk}(D) = \mathcal{S}(D)$*. That means return all tuples of* $\mathcal{S}^0_{ml}(D)$*. 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:*

$$\left|\bigcup_{i=0}^{j-1} \mathcal{S}^i_{ml}(D)\right| < k \leq \left|\bigcup_{i=0}^{j} \mathcal{S}^i_{ml}(D)\right| \tag{7}$$

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

B. The Top-k Lattice Skyline Algorithm (TkLS)

In this section, we adapt the concept of *multi-level Skyline* computation in Section III to the computation of *top-k Skylines*.

Algorithm 3 returns a set of all $\mathcal{S}^i_{ml}$ sets, hence the first k elements of these sets correspond to the *top-k* elements. However, in a top-k approach it is not necessary to compute all strata. To return the correct number of results, we will loop through the different $\mathcal{S}^i_{ml}$ sets in order of their level and keep the sum of tuples belonging to them. We have to find the $\mathcal{S}^i_{ml}$ sets that completely belong to the top-k results. That means, it is enough to compute l multi-levels such that

$$k \leq \left|\bigcup_{i=0}^{l} \mathcal{S}^i_{ml}(D)\right| \tag{8}$$

From an algorithmic point of view we adjust the multi-level Skyline algorithm as follows: Instead of dealing with node states like *dominated*, *non-empty*, or *empty* in the *Adding Phase* (cp. Section II-B), each node in the BTG is represented by an integer counter, counting the number of tuples belonging to the node. During the adding phase, this counter is increased. In addition, for each level of $\mathcal{S}^i_{ml}$, we keep track of the number of tuples belonging to it. Each time the $\mathcal{S}^i_{ml}$ set level i for a non-empty node is determined, the number of tuples belonging to this $\mathcal{S}^i_{ml}$ set is increased by the number of tuples belonging to the node. When all tuples are read, the classification is done just as described in Section III-B, Algorithm 3, but we append a simple break condition after line 19, which checks the above equation. Afterward, we can return the top-k elements.

C. Examples

In this section we provide some examples for top-k Skyline query computation and discuss different strategies for returning result objects.

Example 7. *Consider a Skyline query on three attributes over the domain* $[2; 2; 1]$*. The lattice structure is given in Figure 5. The small index numbers next to each node show the number of tuples represented by each node. Nodes without index are empty.*

After reading all input objects and classifying the nodes, k *tuples should be returned. The different* $\mathcal{S}^l_{ml}$ *set levels and their sizes can be found in Table II. We will see what happens for different values of* k*. The three cases correspond to those described in Definition 6.*

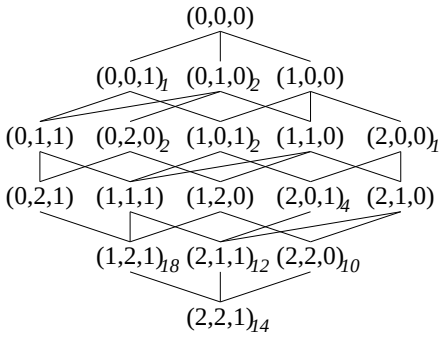


Figure 5. BTG for Example 7.

TABLE II. Nodes and $\mathcal{S}^l_{ml}$ set levels.

l	0	1	2	3	4
nodes in $\mathcal{S}^l_{ml}(D)$	(0,0,1) (0,1,0) (2,0,0)	(0,2,0) (1,0,1)	(2,0,1) (1,2,1) (2,2,0)	(2,1,1)	(2,2,1)
$\lvert\mathcal{S}^l_{ml}(D)\rvert$	4	4	32	12	14

1) *k = 3: Three of four tuples belonging to the nodes of* $\mathcal{S}^0_{ml}(D)$ *are returned.*
2) *k = 4:* $\mathcal{S}^0_{ml}(D)$ *is returned.*
3) *k = 10:* $\mathcal{S}^0_{ml}(D)$ *and* $\mathcal{S}^1_{ml}(D)$ *are returned completely, leading to 8 tuples in the result set. Additionally,* $k-8=2$ *tuples from* $\mathcal{S}^2_{ml}(D)$ *are returned.*

Please note that the proposed algorithm will not return tuples in a progressive way. A tuple with a higher overall level than another could be returned, just because of the order of the input relation. The next example will outline such a scenario:

Example 8. *Reconsider the Skyline query from Example 7. The* top-1 *result should be returned. Tuples belonging to the corresponding nodes are read in the following order:*

$$(0,2,0),\ (2,1,1),\ (2,0,0),\ (0,1,0),\ \ldots$$

The third tuple read is the first one in $\mathcal{S}^0_{ml}(D)$*. As the top-1 query is looking for only one result, the algorithm will stop after reading (and returning)* $(2,0,0)$.

One may criticize that picking the *top-k* results from the different equivalence classes (nodes of the BTG) is arbitrary in some manner, especially if a multi-level set $\mathcal{S}^j_{ml}$ only partially belongs to the top-k result set. In most cases, there will be *one* $\mathcal{S}^j_{ml}$ set only partially belonging to the top-k results. From $\mathcal{S}^j_{ml}$ only $k - \left|\bigcup_{i=0}^{j-1} \mathcal{S}^i_{ml}(D)\right|$ tuples have to be returned such that the total number of k results is matched. All other tuples are discarded. In this case we have to pick some arbitrary elements out of this $\mathcal{S}^j_{ml}$ set to fill up the top-k elements.

To handle this *"problem"* we can think about some kind of ordering or sorting before returning the top-k elements. Only those tuples belonging to some specific equivalence classes could be returned. The information on the number of tuples in the different equivalence classes may as well be another criteria. By using this information, a number of different top-k queries could be executed with only a small need for computations. An additional weighting of the Skyline attributes can be used to sort the nodes differently. Still, information on the $\mathcal{S}^l_{ml}$ set level of a node can be used, taking it as some attribute result candidates are ordered by. Whichever additional conditions and characteristics are used, the top-k results can be taken then from the nodes coming first in the new order, as Example 9 shows.

Example 9. *After the computations for answering the query in Example 7, some kind of presentation preference may induce a different weighting of the attributes. Imagine that the first attribute is more important than the second, and the second one is more important than the third. Such presentation preferences often are added to user preferences in online shops [27].*

To answer this query, the set of non-empty nodes in the BTG has to be identified. Then, these nodes are ordered ascending w.r.t. the level value for the first attribute. Nodes with equal level value are ordered ascending w.r.t. their $\mathcal{S}^l_{ml}$ *set level* l*. For the non-empty nodes of the BTG, this leads to the following order w.r.t. the query (with best elements being on top). Nodes pooled in a set* $\{,\}$ *remain unordered, due to Skylines being strict partial orders [12].*

$$\begin{gathered}\{(0,0,1),\ (0,1,0)\},\\ \{(0,2,0)\},\\ \{(1,0,1)\},\\ \{(1,2,1)\},\\ \{(2,0,0)\},\\ \{(2,0,1),\ (2,2,0)\},\\ \{(2,1,1)\},\\ \{(2,2,1)\}\end{gathered}$$

The nodes holding the top-k tuples than can now be identified by suming up the number of tuples belonging to each of the nodes. Using the values of Example 7 (and Figure 5), a top-5 query could be answered by returning all tuples of the nodes $(0,0,1)$*,* $(0,1,0)$*, and* $(0,2,0)$*. Please note that a top-4 query would return all tuples of nodes* $(0,0,1)$ *and* $(0,1,0)$*, but not all of* $(0,2,0)$.

We omit further discussions of the effects of different ordering strategies here as w.r.t. the original Skyline query, all candidates in $\mathcal{S}^j_{ml}$ are equally good results. We have seen that in spite of being developed for Skyline queries, our *multi-level Skyline algorithm* can easily be applied to *top-k Sykline queries* as well. Its greatest advantage remains: the *linear* runtime complexity in the number of input tuples and size of the BTG.

V. External Lattice

All lattice-based algorithms (e.g., [10][11][20]) keep the complete lattice structure in main memory. Hence all nodes must be in memory, may it be in an array, a hash map, or some other data structure. Following [20], the memory requirements is linear w.r.t. the size of the BTG:

$$mem(BTG) = \left\lceil \frac{1}{4} \prod_{i=1}^{m} (\max(A_i)+1) \right\rceil \quad \text{(in bytes)}$$

where $\max(A_i)$ is the maximal value of attribute A_i in the low-cardinality domain $\mathrm{dom}(A_1) \times \ldots \mathrm{dom}(A_m)$, cp. Definition 4.

However, memory requirements can be very high in some cases. The logical next step is to develop external algorithms for multi-level and top-k Skyline computation.

The idea behind our approach is to always keep one level of the BTG in memory. We need each node together with the information if a tuple belongs to it and its $\mathcal{S}^i_{ml}$. For a BTG level c, we read through the input relation and mark each node we find a tuple belonging to it as *non-empty*. After all tuples have been read in the current round, we compute the next level $c+1$, with all nodes marked *empty*. We then walk through the nodes in level c and for each node n we set the $\mathcal{S}^i_{ml}$ of the dominated nodes to

$\mathcal{S}^i_{ml}$ iff n is *empty* and
$\mathcal{S}^{i+1}_{ml}$ iff n is *non-empty*.

After this, the $\mathcal{S}^i_{ml}$ of level c can be written to external memory and removed from main memory. During this "information handover" to the next level, we need to keep two levels in memory. We need the level we just were working with to deliver information of domination to the next level. As we have to be able to deal with the BTG's levels with the most nodes, we require enough main memory to store the level with the most nodes twice. To analyze this amount, we need some information on the structure of the lattice (see [22]).

Theorem 1. *For a BTG (lattice) over a d-dimensional Skyline query on the domain* $\mathrm{dom}(A) = \mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_d)$ *the following holds:*

a) Height of the BTG:

$$height(BTG) = 1 + \sum_{i=1}^{d} \max(A_i) \tag{9}$$

b) Width of the BTG at a specific level l, $width(BTG, l)$*:*

$$width(BTG, l) \quad = \quad w(l, \{A_1, \ldots, A_d\}), \text{ where}$$

$$w(l, \{A_1, \ldots, A_d\}) = \sum_{i=0}^{\min(l, \max(A_1))} w(l - i, \{A_2, \ldots, A_d\}), \quad \text{if } d > 1$$

$$w(l, \{A_d\}) \quad = \quad \begin{cases} 0, & \text{if } \max(A_d) < l \\ 1, & \text{if } \max(A_d) \geq l \end{cases}$$

c) Maximum width of the BTG:

Due to the symmetrical structure of a BTG, the level with the maximum width will occur at level

$$\lfloor \frac{height(BTG)}{2} \rfloor$$

Note that an efficient algorithm for computing the width of the lattice for a given Skyline query can be found in [28].

As we have to work from level to level, it is very useful to be able to compute a *first node* for a given level l without any information about nodes in other levels. To create a given level we start with the left-most node. The level value for each A_j of the left-most node of a BTG for a Skyline query over $\mathrm{dom}(A) := \mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_d)$ can be found using the following expression:

$$\max(\min(l - \sum_{i=j+1}^{d} \max(A_i), \max(A_j)), 0) \tag{10}$$

With this formula we set the level values of A_d to the highest possible value for the given level l. Then we set the level value for A_{d-1} to $l - \max(A_d)$ (if possible). If $l - \max(A_d) > \max(A_{d-1})$, parts of the level sum l are used to "fill" the level value at position $d-2$ and so on.

Example 10. *Consider Figure 3. The first node of level* 3 *for example is computed as follows:*

- *for* A_1*:* $\max(\min(3-4, 2), 0) = 0$
- *for* A_2*:* $\max(\min(3, 4), 0) = 3$

Therefore, we have the node $(0, 3)$.

Algorithm 4 is a more generic version of this formula that can do the filling with only for just a part of the A_i in $\mathrm{dom}(A)$. Applying Algorithm 4 to all A_i obviously yields equal results as above Equation (10) and produces the left-most node of a BTG w.r.t. the BFT.

Algorithm 4 FILL-UP

Input: node n, $index$, $level\ l$, Skyline query over $\mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_d)$
Output: a BTG node

```
1:  function FILL-UP(n, index, l)
2:      dist ← 0
3:      x ← n
4:      // loop over the A_i
5:      for i ← d, ..., index do
6:          // apply Equation (10)
7:          x[i] := max(min(l − Σ_{j=i+1}^{d} x[j], max(A_i)), 0)
8:          dist ← dist + x[i]
9:      end for
10:     if dist ≠ 0 then
11:         // no valid distribution can be found
12:         return NIL
13:     end if
14:     return x
15: end function
```

To find the next node y in the BFT for a given node $x := (x_1, \ldots, x_d)$, we shift one level value to a position more left (i.e., with a lower index i) and fill the rest of the node with Algorithm 4. This shift function is given in Algorithm 5. It is repeatedly used to create a whole level of the BTG in Algorithm 6. With all these helping procedures, we can define Algorithm 7, *External TkLS*.

Further optimizations are straightforward. As during the switch to the next level we have to keep two levels in memory, we can always keep at least two levels in memory and process tuples of all nodes of these. We actually are able to hold a higher number of levels in memory at most times. Starting at level 0, in the first step we can compute as many levels as we can fit into memory completely. Then the reading of the input relation can be done for multiple levels at once. The level switch then has to be adjusted to compute the $\mathcal{S}^i_{ml}$ of all nodes in memory, write those to external memory, and remove the information from main memory. Before the highest BTG

Algorithm 5 NEXT-NODE

Input: node n, Skyline query over $\mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_d)$
Output: next node in same level in BFT search

```
1:  function NEXT-NODE(n)
2:      l ← Σ_{i=1}^{d} n[i]
3:      for i ← d, ..., 1 do
4:          if n[i] > 0 then
5:              n[i] = n[i] − 1
6:              for j ← i − 1, ..., 1 do
7:                  if n[j] < max(A_j) then
8:                      n[j] = n[j] + 1
9:                      rem ← l − Σ_{k=1}^{j} n[k]
10:                     return FILL-UP(n, 1, l - rem)
11:                 end if
12:             end for
13:         end if
14:     end for
15:     // no more node in current level can be found
16:     return NIL
17: end function
```

Algorithm 6 CREATE-LEVEL

Input: level l, Skyline query over $\mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_d)$
Output: a list of nodes of the given level

```
1:  function CREATE-LEVEL(l)
2:      // init a list to hold nodes
3:      result ← list()
4:      node ← init node of level 0
5:      node ← FILL-UP(node,1,l)
6:      while (node ≠ NIL) do
7:          result.add(node)
8:          node ← NEXT-NODE(node)
9:      end while
10:     // no more nodes in current level can be found
11:     return result
12: end function
```

level is processed, the next level has to be calculated - or more than one level, if they fit into memory. That way any amount of memory between the minimum requirement and enough to keep the whole BTG in main memory can be used efficiently. If more than two levels fit into memory in average, the number of loops through the input relation is lower than for one level in memory. The algorithm will actually turn into a variant of in-memory *TkLS* when the whole BTG fits into memory and the input relation has to read only once.

Please note that if working with more than one level at a time, we need to loop through the input relation twice for every set of levels in memory. In the first loop, we only mark non-empty nodes. After this first loop, we can do a BFT/DFT to find the correct $\mathcal{S}^i_{ml}$ for the nodes in memory. The second loop through the input relation will then return the tuples together with their $\mathcal{S}^i_{ml}$. Tuples belonging to the lowest level in memory could be returned instantly in the first loop as for them the $\mathcal{S}^i_{ml}$ will not change anymore. As we work on at least two levels at a time, the number of loops through the input relation is at most as high as while working on only one level (i.e., $\max(A)$ loops). For the lowest (and highest) levels, potentially more than two levels fit into memory.

Algorithm 7 External TkLS

Input: Skyline query over $\mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_d)$

```
1:  prvLvl ← NIL
2:  for i ← 0, ..., Σ_{i=1}^{d} max(A_i) do
3:      currLvl ← CREATE-LEVEL(i)
4:      // walk through prvLvl and set S^i_ml for
5:      // all nodes in currLvl
6:      // read input relation and
7:      // • mark non-empty nodes in currLvl
8:      // • return each input tuple with its S^i_ml
9:      prvLvl ← currLvl
10: end for
```

Memory requirements and worst case performance of our algorithms with an external BTG can be found in Lemma 5.

Lemma 5 (Properties of External TkLS)**.**

a) The memory requirements for $TkLS$ with an external lattice is given by:

$$mem(BTG) := 2 \cdot \max(width(BTG)) \tag{11}$$

b) The number of rounds the input relation has to be read is given by:

$$rounds := n \cdot height(BTG) \tag{12}$$

c) The runtime complexity is given by the number of rounds, hence

$$\mathcal{O}(n \cdot height(BTG)) \tag{13}$$

We will see *External TkLS* at work in Example 11.

Example 11. *We will apply* External TkLS *to a Skyline query and an input dataset D resembling the ones known from Example 6 and Figure 4. Our main memory is big enough to hold information of 6 nodes (twice the width of the BTG).*

Round 1:

- *$currLvl$ is level 0 with node $(0,0)$.*
  The node is marked as belonging to $\mathcal{S}^0_{ml}$.
- *Input relation is read, but no tuples are found.*

Round 2:

- *$currLvl$ is level 1 with nodes $(0,1)$ and $(1,0)$.*
  All nodes are marked as belonging to $\mathcal{S}^0_{ml}$.
- *Input relation is read.*
  Node $(1,0)$ is non-empty.
  Tuples belonging to it are returned as part of $\mathcal{S}^0_{ml}$.

Round 3:

- *$currLvl$ is level 2 with nodes $(0,2)$, $(1,1)$, and $(2,0)$.*
  Nodes $(0,2)$ and $(1,1)$ are marked as $\mathcal{S}^1_{ml}$, nodes $(0,2)$ as $\mathcal{S}^0_{ml}$.
- *Input relation is read.*
  Nodes $(1,1)$ and $(2,0)$ are non-empty.
  Tuples are returned as parts of $\mathcal{S}^0_{ml}$ resp. $\mathcal{S}^1_{ml}$.

The algorithm continues until Round 7 (for level 6) in which tuples in $\mathcal{S}^3_{ml}$ are returned.

This algorithm can easily be adjusted to compute a "normal" Skyline. We just have to omit the $\mathcal{S}^i_{ml}$ and only keep the status (*emtpy*, *non-emtpy*, or *dominated*) for each node. Only nodes belonging to *non-emtpy* nodes will be returned as Skyline results. An additional stop criteria could be added as well: The Skyline is found as soon as one level only holds *non-emtpy* and *dominated* nodes.

VI. Remarks

In this section we will discuss some additional points on our lattice-based multi-level and top-k algorithms.

A. High-Cardinality Domains

The lattice based multi-level and top-k algorithms are restricted to low-cardinality domains. Since this is a huge restriction to Skyline computation, we want to adjust our algorithms to handle high-cardinality domains as well. One approach to handle large domains is suggested in [29]. The idea is to use a down-scaling of a high-cardinality domain to a small domain such that a lattice based algorithm can be used as some kind of pre-filtering, which eliminates objects before the final Skyline computation. Afterward, a BNL style algorithm is necessary for final evaluation.

This approach is in focus when handling large domains for multi-level and top-k Skyline computation. Instead of eliminating objects in the lattice-based pre-filtering phase, a similar approach as in Section IV might be used. The down-scaling [29] leads to several comparable objects in the scaled equivalence classes. An additional counter for each equivalence class as well as an overall counter for each level in the scaled lattice could serve as a break condition similar to our top-k Skyline algorithm. Afterward, a BNL-style top-k algorithm like EBNL or ESFS (cp. [25]) could do the final computation of the k best objects.

B. Parallelization

Since multi-core processors are going mainstream, one might also think about a parallel variant of TkLS. For the first two phases, the *Construction Phase* and the *Adding Phase* (cp. Section II-B) we can follow the approach of [20], which presents different parallel implementations of the lattice-based algorithms. They also show how to parallelize phase 3, the *Removal Phase*, in Lattice Skyline. However, since in TkLS the removal phase is replaced by a *Classification Phase*, the proposed parallelization does not apply anymore. The parallelization of the classification phase shown in Algorithm 3 is not straightforward, since line 12 – where the temporary value tmp_{ml} is computed – depends on some pre-computations, which can only be done following sequential execution. Therefore, parallelization of TkLS is restricted to phase 1 and phase 2. However, following [20], the adding phase takes the most computation time. Hence, we expect an enormous speed-up in multi-level and top-k Skyline computation when applying phase 2 in parallel.

VII. Experiments

This section provides our benchmarks on synthetic and real data to reveal the performance of the outlined algorithms.

A. Benchmark Framework

The concept of MLLS is the basis of TkLS, hence the performance of our top-k approach reflects the power of our multi-level Skyline algorithm. And since there is no competitor for MLLS, we only compared our approach *TkLS* to the state-of-the-art algorithms in generic top-k Skyline computation, *Extended Block-Nested-Loop* (EBNL) and *Extended Sort-Filter-Skyline* (ESFS) [25]. EBNL is a variant of the standard BNL algorithm [4] with the modification that each computed stratum is removed from the dataset and the Skyline is computed again. ESFS is an extension of SFS [8] exploiting some kind of data pre-sorting. In the worst-case EBNL and ESFS have a time complexity of $\mathcal{O}(n^2)$, whereas all lattice based algorithms have linear runtime complexity. Note that there are other top-k Skyline algorithms (cp. Section VIII), but all of them exploit some index structure. Since our algorithm is generic for all kind of input data, we do not compare index based algorithms to our approach. Also note that we used the non-external version of our algorithm, because EBNL and ESFS are implemented as main memory algorithms.

All algorithms have been implemented in Java 7.0. TkLS follows the implementation details given in [20] and [10]. The experiments were performed on a machine running Debian Linux 7.1 equipped with an Intel Xeon 2.53 GHz processor. Our prototype is available as open source project on GitHub *https://github.com/endresma/TopKSkyline.git*.

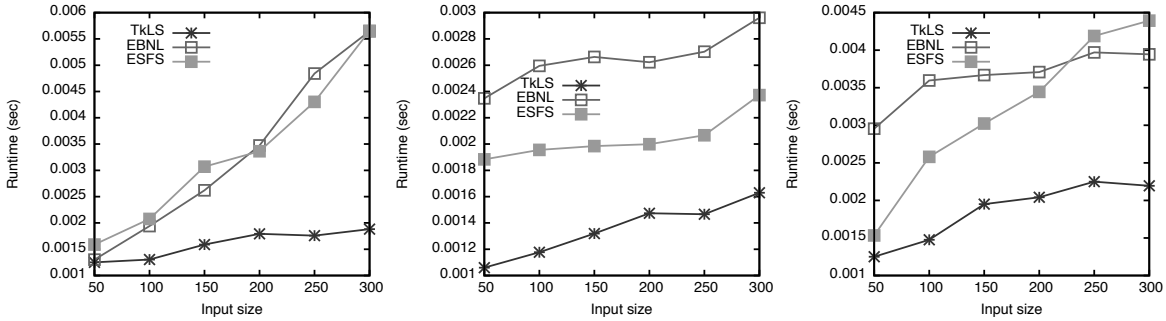
For our synthetic datasets we used the data generator commonly used in Skyline research [4]. We generated *anti-correlated* (anti), *correlated* (corr), and *independent* (ind) distributions and varied (1) the data cardinality n, (2) the data dimensionality d, and (3) the top-k value. For the experiments on real-world data, we used the well-known *Zillow* and *NBA* datasets. All datasets where adapted for low-cardinality Skyline computation. The Zillow dataset crawled from *www.zillow.com* contains more than 2M entries about real estate in the United States. Each entry includes number of *bedrooms* and *bathrooms*, *living area* in sqm, and *age* of the building. The NBA dataset is a small 5-dimensional dataset containing 17264 tuples, where each entry records performance statistics for a NBA player. Following [30], NBA is a fairly correlated dataset. Both datasets serve as real-world applications, which require finding the Skyline on data with a low-cardinality domain.

B. Experimental Results

We now present our experimental results.

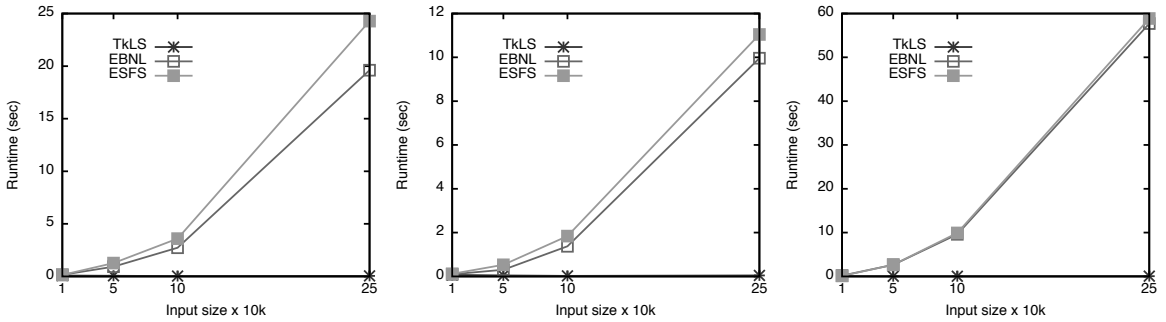
Figure 6 presents the behavior of our TkLS algorithm on a typical top-k Skyline query. We chose a 3-dimensional dataset because this is very common in Skyline or Pareto preference selection. The low-cardinality domain was constructed by $[3; 4; 4]$, which might correspond to attributes like "board" (none, breakfast, half board, full board), "star ratings" (1*–5*), and "price ranges" (e.g., $[40-80[$, $[80-120[$, $[120-160[$, $[160-200[$, $[200-240[$) in a search for the best hotel. To keep the hotel example we chose $k = 5$ (a user wants only to see a few hotels) and restricted the input size to 50 to 300 objects.

We present the runtime on anti-correlated, correlated, and independent data. Figure 6a shows the result on anti-correlated data. TkLS clearly outperforms EBNL and ESFS even though TkLS has some overhead in constructing the lattice. ESFS and



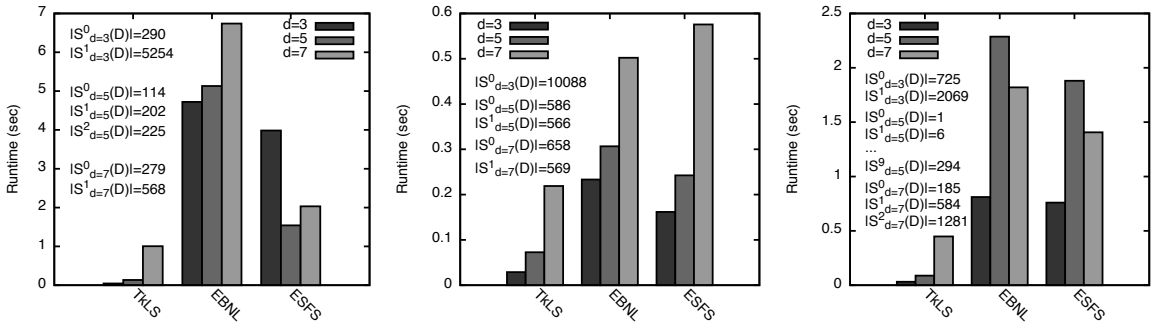
(a) Runtime: anti-correlated data. (b) Runtime: correlated data. (c) Runtime: independent data.

Figure 6. Runtime results on small input sizes: $d = 3$, top-5.



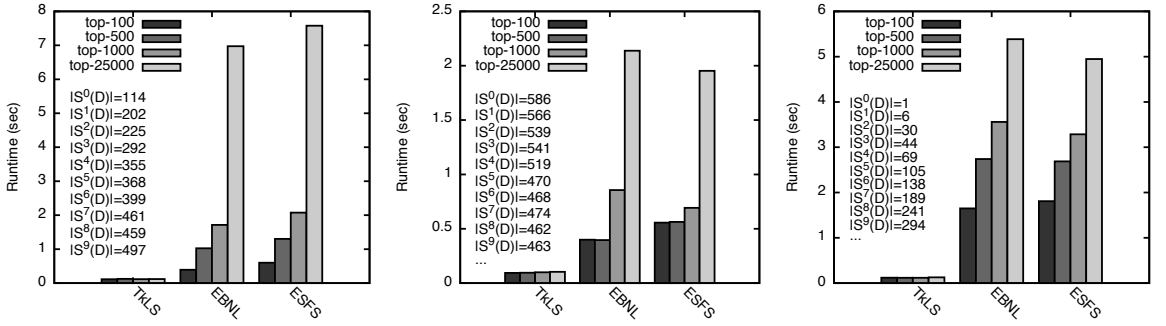
(a) Runtime: anti-correlated data. (b) Runtime: correlated data. (c) Runtime: independent data.

Figure 7. Runtime results on different input sizes: $d = 5$, top-500.



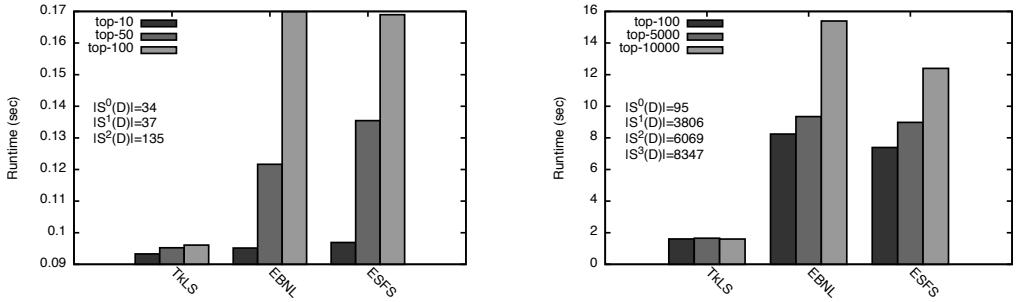
(a) Runtime: anti-correlated data, top-500. (b) Runtime: correlated data, top-1000. (c) Runtime: independent data, top-1000.

Figure 8. Experimental results on different dimensions: $d = 3, 5, 7$, $n = 50K$.



(a) Runtime: anti-correlated data. (b) Runtime: correlated data. (c) Runtime: independent data.

Figure 9. Influence of different k values: $k \in \{100, 500, 1K, 25K\}$, $d = 5$, $n = 50K$.



(a) NBA data: $d = 5$, $n = 17264$. (b) Zillow data: $d = 4$, $n = 2M$.

Figure 10. Experimental results on real data.

EBNL are almost equally good. Note that $|\mathcal{S}^0_{ml}(D)| = 7$, that means the top-5 objects lie in the first stratum, i.e., the Skyline set. In the case of correlated (Figure 6b) and independent (Figure 6c) ESFS outperforms EBNL, but is worse than TkLS. Therefore, our multi-level algorithm is the first choice when evaluating Skyline queries on small datasets and low-cardinality domains.

Figure 7 presents the runtime of all algorithms on a 5-dimensional anti-correlated, correlated, and independent distributed dataset. We used a *top-500* query on different data cardinality. The low-cardinality domain was constructed by $[2; 3; 5; 10; 100]$. For all Skyline sets it holds that $|S(D)| < 500$ to get the effect of computing more than the 0-stratum. In contrast to the previous experiment we used larger input sizes (10K to 250K). Here, TkLS clearly outperforms EBNL and ESFS, which rely on a tuple-to-tuple comparison. TkLS always constructs the same lattice for all kind of data, and hence, has similar runtimes for all kind of input sizes.

Figure 8 shows the runtime of all algorithms on 3, 5, and 7 dimensions having anti-correlated, correlated, and independent data (up to $[2; 3; 5; 10; 10; 10; 100]$). The underlying data cardinality is $n = 50000$ and the target was to find the *top-500* respectively the *top-1000* elements. We also present the size of the different multi-level Skylines. For example, in the anti-correlated data (Figure 8a), if $d = 3$ we have $|\mathcal{S}^0_{ml}(D)| = 290$ and the first stratum has $|\mathcal{S}^1_{ml}(D)| = 5254$ objects. This is also the reason why ESFS in this case is worse than for $d = 5$ or $d = 7$. ESFS has to compare all objects of the first stratum to all others, not yet dominated tuples. Figure 8b and 8c show the results on correlated and independent datasets. In all our benchmarks TkLS outperforms EBNL and ESFS due its small lattice structure and independency of the data distribution, cp. [11].

Figure 9 visualizes the effect of different values of k for anti-correlated, correlated, and independent data distributions. Therefore we computed top-k elements for $k \in \{100, 500, 1K, 25K\}$ using 5 dimensions (as in Figure 7) and a data cardinality of $n = 50000$.

The runtime for TkLS for all ks and all distributions is very similar. This is due to the lattice based approach, where no tuple-to-tuple comparison is necessary, but only the construction of the BTG. Since the BTG for all ks is the same, the runtime for all top-k queries is quite similar.

Considering the results on anti-correlated data in Figure 9a, in the top-100 query only the Skyline $S(D)$ has to be computed. For $k = 500$ we have to compute stratum 0, 1, and 2. For top-1000 the first five strata are necessary, and for $k = 25K$ we need 41 strata to answer the query. We also see in this experiment that ESFS exploiting some pre-sorting is worse than EBNL. This is due the reordering of the elements. The results for correlated (Figure 9b) and independent data (Figure 9c) are quite similar. For the correlated and independent data we decided to use $k = 1000$ such that at least the second stratum must be computed to fulfill the 1000 objects.

Figure 10a shows the results on the NBA dataset, where the domain is drawn from $[10; 10; 10; 10; 10]$. The NBA set has a size of 17265 objects. Again, TkLS has similar runtime for each k whereas the runtime for EBNL and ESFS increases with larger k values.

In **Figure 10b** we present the Zillow dataset (domain $[10; 10; 36; 46]$). We compute the top-k elements for $k \in \{100, 5000, 10000\}$ to show the effect of computing different strata. TkLS clearly outperforms EBNL and ESFS. Again, since our algorithm is based on the lattice of a Skyline query the runtimes for the different ks are quite similar. EBNL and ESFS have to compute four strata to fulfill the $k = 10000$ query, which results in a long runtime and hence bad performance.

VIII. Related Work

The most prominent algorithms for Skyline computation are based on a block-nested-loop style tuple-to-tuple comparison (e.g., BNL [4]). Based on this several algorithms have been published, e.g., the NN algorithm [6], SFS [8], or LESS [9], to just list some. Many of these algorithms have been adapted for parallel Skyline computation, e.g., [17][19][31]. There are also algorithms utilizing an index structure to compute the Skyline, e.g., [7][32]. 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* [11] and *Hexagon* [10], both having a *linear time complexity*. There is also work on parallel preference computation exploiting the lattice structure [20]. The authors of [29] present how to handle high-cardinality domains and therefore makes lattice algorithms available for a broad scope of applications.

The present paper is an extended version of [1], where the basics of multi-level and top-k Skylines were discussed.

The idea of *multi-level Skylines* was already mentioned by Chomicki [12] under the name of *iterated preferences*. However, Chomicki has never presented an algorithm for the computation of multi-level preferences.

In [33] the term "multi-level Skyline" was used, but it has nothing to do with the objects behind the Skyline. It was only used for a Skyline algorithm where a pre-computed Skyline dataset was used to compute continuous Skylines. Apart from that there is no other work on computing the i-th stratum of a Skyline query.

Regarding top-k [23][34] and Skyline [3] queries, there are some approaches that combine these both paradigms to *top-k Skyline queries*. In [24] and [25] the authors calculate the first stratum of the *Skyline* with some sort of post-processing. Afterward, they define the k best objects or continue Skyline computation without the first stratum. A similar approach was followed in [35], which presented the first study for processing top-k dominating queries over distance-based dynamic attribute vectors defined over a metric space. They present the Skyline-Based Algorithm (SBA) to compute the top-1 dominating object, which is removed from the dataset and the same process is repeated until all top-k results have been reported. The authors of [26] abstract Skyline ranking as a dynamic search over Skyline subspaces guided by user-specific preferences. In [36][37][38] and [39] 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. [40] consider top-k combinatorial Skyline queries, and Zhang et al. [41] discuss a probabilistic top-k Skyline operator over uncertain data. Top-k queries are also of interest in the computation of spatial preferences [42][43], where the aim is to retrieve the k best objects in a spatial neighborhood of a feature object. Yu et al. [44] consider the problem of processing a large number of continuous top-k queries, each with its own preference. The authors of [45] present a framework for top-k query processing in large-scale P2P networks, where the dataset 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.

Although there is some related work, the problem of efficiently evaluating top-k Skylines is still an open issue.

IX. Conclusion and Future Work

In this paper we discussed the iterated evaluation of a Skyline query. For this, we provided a deep insight into lattice-based Skyline algorithms, and afterward presented how to modify Lattice Skyline to get multi-level Skyline sets. After a running through a set of input tuples, we are able to return not only the Pareto frontier, but also the tuples that are directly dominated by them, and so on. Our approach supports multi-level and top-k Skyline computation without computing each stratum of the Skyline query individually or relying on the time-consuming tuple-by-tuple comparisons. Comprehensive experiments show the benefit of our approach in comparison to existing techniques. Furthermore, the external version of our algorithm allows the evaluation of top-k Skyline queries even when main memory is low. For future work we want to get rid of the restriction on low-cardinality domains. Therefore, we want to adjust our algorithms as suggested in Section VI-A. However, this could be a challenging task.

References

[1] M. Endres and T. Preisinger, "Behind the Skyline," in Proceedings of DBKDA '15: The 7th International Conference on Advances in Databases, Knowledge, and Data Applications. IARIA, 2015, pp. 141–146.

[2] K. Stefanidis, G. Koutrika, and E. Pitoura, "A Survey on Representation, Composition and Application of Preferences in Database Systems," ACM Transaction on Database Systems, vol. 36, no. 4, 2011.

[3] J. Chomicki, P. Ciaccia, and N. Meneghetti, "Skyline Queries, Front and Back," Proceedings of SIGMOD '13: ACM SIGMOD International Conference on Management of Data, vol. 42, no. 3, 2013, pp. 6–18.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker, "The Skyline Operator," in Proceedings of ICDE '01: International Conference on Data Engineering. Washington, DC, USA: IEEE, 2001, pp. 421–430.

[5] S. Mandl, O. Kozachuk, M. Endres, and W. Kießling, "Preference Analytics in EXASolution," in Proceedings of BTW '15: Datenbanksysteme für Business, Technologie und Web, 2015.

[6] D. Kossmann, F. Ramsak, and S. Rost, "Shooting Stars in the Sky: An Online Algorithm for Skyline Queries," in Proceedings of VLDB '02: 28th International Conference on Very Large Data Bases, pp. 275–286.

[7] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An Optimal and Progressive Algorithm for Skyline Queries," in Proceedings of SIGMOD '03: Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM, 2003, pp. 467–478.

[8] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with Presorting," in Proceedings of ICDE '03: International Conference on Data Engineering, 2003, pp. 717–816.

[9] P. Godfrey, R. Shipley, and J. Gryz, "Algorithms and Analyses for Maximal Vector Computation," The VLDB Journal, vol. 16, no. 1, 2007, pp. 5–28.

[10] T. Preisinger and W. Kießling, "The Hexagon Algorithm for Evaluating Pareto Preference Queries," in Proceedings of MPref '07: Multidisciplinary Workshop on Preference Handling, 2007.

[11] M. Morse, J. M. Patel, and H. V. Jagadish, "Efficient Skyline Computation over Low-Cardinality Domains," in Proceedings of VLDB '07: International Conference on Very Large Data Bases, 2007, pp. 267–278.

[12] J. Chomicki, "Preference Formulas in Relational Queries," in TODS '03: ACM Transactions on Database Systems, vol. 28, no. 4. New York, NY, USA: ACM Press, 2003, pp. 427–466.

[13] W. Kießling, "Foundations of Preferences in Database Systems," in Proceedings of VLDB '02: International Conference on Very Large Data Bases. Hong Kong, China: VLDB, 2002, pp. 311–322.

[14] W. Kießling, M. Endres, and F. Wenzel, "The Preference SQL System - An Overview," Bulletin of the Technical Commitee on Data Engineering, IEEE Computer Society, vol. 34, no. 2, 2011, pp. 11–18.

[15] P. Fishburn, "Preference Structures and their Numerical Representation," Theoretical Computer Science, vol. 217, no. 2, 1999, pp. 359–383.

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

[17] J. Selke, C. Lofi, and W.-T. Balke, "Highly Scalable Multiprocessing Algorithms for Preference-Based Database Retrieval," in Proceedings of DASFAA '10: International Conference on Database Systems for Advanced Applications, ser. LNCS, vol. 5982. Springer, 2010, pp. 246–260.

[18] S. Park, T. Kim, J. Park, J. Kim, and H. Im, "Parallel Skyline Computation on Multicore Architectures," in Proceedings of ICDE '09: International Conference on Data Engineering. Washington, DC, USA: IEEE Computer Society, 2009, pp. 760–771.

[19] S. Liknes, A. Vlachou, C. Doulkeridis, and K. Nørvåg, "APSkyline: Improved Skyline Computation for Multicore Architectures," in Proceedings of DASFAA '14: International Conference on Database Systems for Advanced Applications, 2014, pp. 312–326.

[20] M. Endres and W. Kießling, "High Parallel Skyline Computation over Low-Cardinality Domains," in Proceedings of ADBIS '14: East European Conference in Advances in Databases and Information System. Springer, 2014, pp. 97–111.

[21] B. A. Davey and H. A. Priestley, Introduction to Lattices and Order, 2nd ed. Cambridge, UK: Cambridge University Press, 2002.

[22] T. Preisinger, W. Kießling, and M. Endres, "The BNL++ Algorithm for Evaluating Pareto Preference Queries," in Proceedings of MPref '06: Multidisciplinary Workshop on Preference Handling, pp. 114–121.

[23] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A Survey of Top-k Query Processing Techniques in Relational Database Systems," ACM Comput. Surv., vol. 40, no. 4, Oct. 2008, pp. 11:1–11:58.

[24] M. Goncalves and M.-E. Vidal, "Top-k Skyline: A Unified Approach," in OTM Workshops, 2005, pp. 790–799.

[25] C. Brando, M. Goncalves, and V. González, "Evaluating Top-k Skyline Queries over Relational Databases," in Proceedings of DEXA '07: 18th International Conference on Database and Expert Systems Applications, ser. LNCS, vol. 4653. Springer, 2007, pp. 254–263.

[26] J. Lee, G.-W. You, and S.-W. Hwang, "Personalized Top-k Skyline Queries in High-Dimensional Space," Information Systems, vol. 34, no. 1, Mar. 2009, pp. 45–61.

[27] S. Fischer, W. Kießling, and T. Preisinger, "Preference based Quality Assessment and Presentation of Query Results," in G. Bordogna, G. Paila: Flexible Databases Supporting Imprecision and Uncertainty, vol. 203. Secaucus, NJ, USA: Springer, 2006, pp. 91–121.

[28] R. Glück, D. Köppl, and G. Wirsching, "Computational Aspects of Ordered Integer Partition with Upper Bounds," in Proceedings of SEA '13: 12th International Symposium on Experimental Algorithms, 2013, pp. 79–90.

[29] M. Endres, P. Roocks, and W. Kießling, "Scalagon: An Efficient Skyline Algorithm for all Seasons," in Proceedings of DASFAA '15: 20th International Conference on Database Systems for Advanced Applications, 2015, in press.

[30] C. Y. Chang, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang, "On High Dimensional Skylines," in Proceedings of EDBT '06: International Conference on Extending Database Technology, ser. LNCS, vol. 3896. Springer, 2006, pp. 478–495.

[31] S. Chester, D. Sidlauskas, I. Assent, and K. S. Bøgh, "Scalable Parallelization of Skyline Computation for Multi-Core Processors," in Proceedings of ICDE '15: International Conference on Data Engineering, 2015, pp. 1083–1094.

[32] K. Lee, B. Zheng, H. Li, and W.-C. Lee, "Approaching the Skyline in Z Order," in Proceedings of VLDB '07: The 33rd International Conference on Very Large Data Bases. VLDB Endowment, 2007, pp. 279–290.

[33] E. El-Dawy, H. M. O. Mokhtar, and A. El-Bastawissy, "Multi-level Continuous Skyline Queries (MCSQ)," in Proceedings of ICDKE '11: International Conference on Data and Knowledge Engineering, IEEE. IEEE, 2011, pp. 36–40, n/a.

[34] M. J. Carey and D. Kossmann, "On saying "Enough already!" in SQL," Proceedings of SIGMOD '97: The ACM SIGMOD International Conference on Management of Data, vol. 26, no. 2, 1997, pp. 219–230.

[35] E. Tiakas, G. Valkanas, A. N. Papadopoulos, and Y. Manolopoulos, "Metric-Based Top-k Dominating Queries," in Proceedings of EDBT '14: The 17th International Conference on Extending Database Technology, 2014, pp. 415–426.

[36] Y. Tao, X. Xiao, and J. Pei, "Efficient Skyline and Top-k Retrieval in Subspaces," IEEE Transactions on Knowledge and Data Engineering, vol. 19, no. 8, 2007, pp. 1072–1088.

[37] M. L. Yiu and N. Mamoulis, "Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data," in Proceedings of the 33rd International Conference on Very Large Data Bases, ser. VLDB '07. VLDB Endowment, 2007, pp. 483–494.

[38] 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, ser. LNCS. Springer, 2009, vol. 5690, pp. 471–485.

[39] P. Pan, Y. Sun, Q. Li, Z. Chen, and J. Bian, "The Top-k Skyline Query in Pervasive Computing Environments," in Joint Conferences on Pervasive Computing (JCPC). IEEE, 2009, pp. 335–338.

[40] I.-F. Su, Y.-C. Chung, and C. Lee, "Top-k Combinatorial Skyline Queries," in Database Systems for Advanced Applications. Berlin, Heidelberg: Springer Berlin Heidelberg, Jan. 2010, pp. 79–93.

[41] Y. Zhang, W. Zhang, X. Lin, B. Jiang, and J. Pei, "Ranking Uncertain Sky: The Probabilistic Top-k Skyline Operator," Information Systems, vol. 36, no. 5, Jul. 2011, pp. 898–915.

[42] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis, "Top-k Spatial Preference Queries," in Proceedings of ICDE '07: 23rd International Conference on Data Engineering. IEEE, 2007, pp. 1076–1085.

[43] 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, vol. 4, no. 2, 2010, pp. 93–104.

[44] A. Yu, P. K. Agarwal, and J. Yang, "Processing a Large Number of Continuous Preference Top-k Queries," in Proceedings of SIGMOD '12: The ACM SIGMOD International Conference on Management of Data. ACM, 2012, pp. 397–408.

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