Behind the Skyline

Markus Endres

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

Timotheus Preisinger

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

***Abstract*—A *Skyline* query selects those tuples from a dataset that are optimal with respect to a set of designated preference attributes. However, in some cases, not only the Pareto frontier is of interest, but also the stratum behind the Skyline. In this paper, we extend the definition of the Skyline to form *multi-level* Skyline sets. We propose an algorithm for multi-level Skyline computation and apply this concept for efficient *top-k Skyline* evaluation. 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. We demonstrate through extensive experimentation on synthetic and real datasets that our algorithm can result in a significant performance advantage over existing techniques.**

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

I. INTRODUCTION

The Skyline operator [1] 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 w.r.t. the natural order $\leq$ in each dimension.

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. 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 both dimensions. 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.

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*. 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 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}.

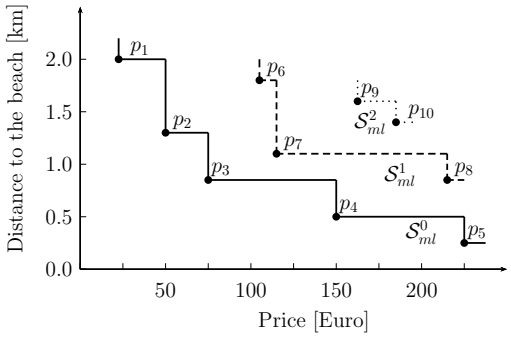


Figure 1. Skyline example.

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 [2]. 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 $|\mathcal{S}| < k$ to 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 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 [2] 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. To our best knowledge, until now there are no efficient algorithms that are specialized on finding *multi-level Skyline* sets or *top-k Skylines*. Motivated by this fact, this paper addresses this issue.

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. We conduct an extensive performance evaluation on synthetic and real datasets in Section V. Section VI contains some related work. Section VII contains our concluding remarks.

 ISBN: 978-1-61208-408-4

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 data set D, i.e., $\mathcal{S}(D)$. Note that Skylines are not restricted to numerical domains [3]. More formally:

Definition 1 (Skyline)**.** *Assume a set of vectors $D \in \mathbb{R}^d$. We define the so called Pareto ordering for all $x = (x_1, ..., x_d)$, $y = (y_1, ..., y_d) \in D$:*

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

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 we prefer the minimal values in each domain and write $x <_\otimes y$ if x is better than y.

In general, algorithms of the block-nested-loop class (BNL) [1] 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. [4]. 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 [4][5] and parallel variants [6][7][8][9] 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. Examples for such algorithms are *Lattice Skyline* [10] and *Hexagon* [11], 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*. This means if $a, b \in D$, the set $\{a, b\}$ has a least upper bound and a greatest lower bound in D. Visualization of such lattices is often done using *Better-Than-Graphs* (*BTG*) [12], 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 considered substitutable.

An example of a BTG over a 2-dimensional space is shown in Figure 2 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. $[2; 4]$). 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. [11]. 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 2, the gray nodes are occupied (*non-empty*) with elements from the dataset whereas the white nodes have no element (*empty*).

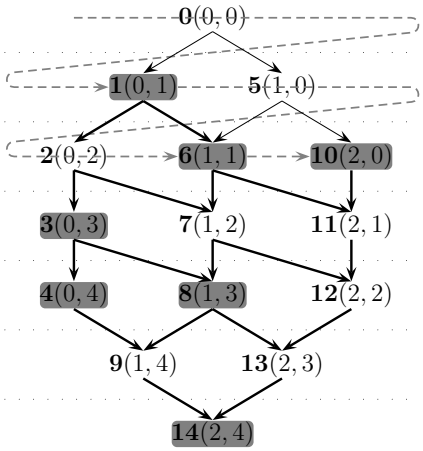


Figure 2. Lattice over $[0..2] \times [0..4]$.

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 2, 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*.

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:

1) **Phase 1:** The *Construction Phase* initializes the data structures. The lattice is represented by an *array* in main memory. 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*.
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 breadth-first order (the dashed line in Figure 2). Each time a *non-empty* and *not dominated* node is found, a DFT will start. 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 2 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.

 ISBN: 978-1-61208-408-4

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

Definition 2 (Multi-Level Skyline)**.** *The multi-level Skyline set of level l (i.e., the l-th stratum) for a dataset D is defined as*

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

Thereby $\mathcal{S}^0_{ml}(D)$ is identical to the standard Skyline $\mathcal{S}(D)$ from Definition 1, and $\mathcal{S}^{l_{max}}_{ml}$ denotes the non-empty set with the highest level.

Lemma 1. *For each tuple t in a finite dataset D, there is exactly one $\mathcal{S}^l_{ml}$ set it belongs to:*

$$\forall t \in D : \left(\exists! l : t \in \mathcal{S}^l_{ml}(D)\right) \tag{4}$$

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

$$\forall y \in \mathcal{S}^l_{ml}(D) : (\exists x \in \mathcal{S}^i_{ml}(D) : x <_{\otimes} y)) \ \ \textit{if} \ \ i < l \tag{5}$$

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 2, $y \in \mathcal{S}^i_{ml}(D)$. 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.

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 1.

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 $\text{tmp}_{ml}(v_i)$ for a node v_i is computed as follows:

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

Algorithm 1 Multi-Level Skyline – Classification Phase

Global data structure: BTG
Output: list of $\mathcal{S}_{ml}$ sets

```
1:  function CLASSIFY
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 1, 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 (line 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 (line 10, same functionality as used in the *removal phase* (DFT) of the lattice Skyline algorithms, cp. [11]). 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 as the equation above (line 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 1. *Figure 3 shows an example of Algorithm 1.*

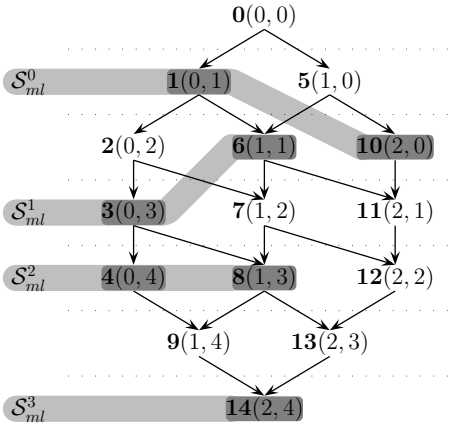


Figure 3. Multi-level Skylines $\mathcal{S}^l_{ml}$.

Since node 0 *is empty, the first relevant node is* 1*. Therefore we set* $\text{tmp}_{ml}[1] = 0$ *and add* 1 *to all direct dominated nodes, i.e.* $\text{tmp}_{ml}[2] = \text{tmp}_{ml}[6] = 1$*. 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* $\text{tmp}_{ml}[3] = \text{tmp}_{ml}[7] = 1$*. Node 6 has already* $\text{tmp}_{ml}[6] = 1$*. The next node is 10, which still has* $\text{tmp}_{ml}[10] = 0$*. Node* 3 *sets* $\text{tmp}_{ml}[4] = \text{tmp}_{ml}[8] = 2$*, and so on. After the BFT has finished we have* 4 $\mathcal{S}_{ml}$ *sets.*

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 [13]. On the other hand, Skyline retrieves tuples where all criteria are equally important concerning some user preference [1]. 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 [14][15].

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 [14][15][16]. 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. More formally:

Definition 3 (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:*

$$\left|\bigcup_{i=0}^{j-1} \mathcal{S}^i_{ml}(D)\right| < k \leq \left|\bigcup_{i=0}^{j} \mathcal{S}^i_{ml}(D)\right| \tag{6}$$

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 1 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. It is enough to compute l multi-levels such that

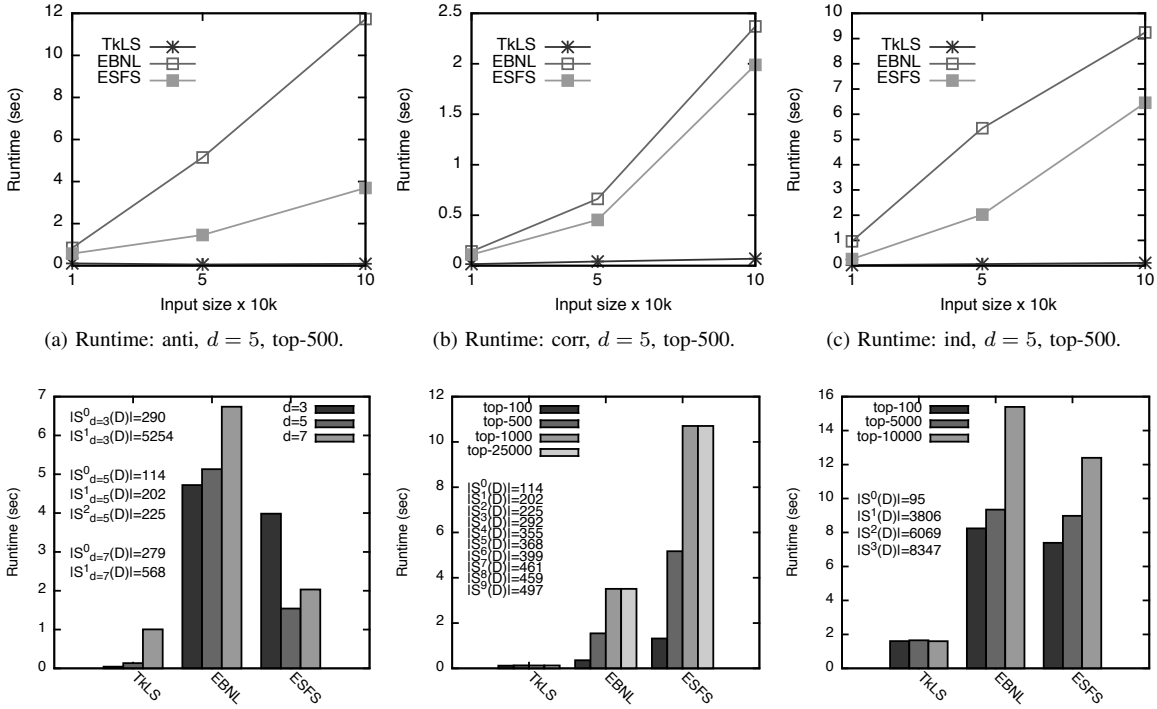
$$k \leq \left|\bigcup_{i=0}^{l} \mathcal{S}^i_{ml}(D)\right| \tag{7}$$

For this, we append a simple break condition after line 19 in Algorithm 1 which checks the above equation. Afterward, we can return the top-k elements.

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 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. Whichever additional conditions and characteristics are used, the top-k results can be taken then from the nodes coming first in the new order. We omit the discussion of the effects of different ordering strategies here as wrt. 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.

 ISBN: 978-1-61208-408-4



(a) Runtime: anti, $d = 5$, top-500. (b) Runtime: corr, $d = 5$, top-500. (c) Runtime: ind, $d = 5$, top-500.

(d) Dimension: anti, $d = 3, 5, 7$, top-500. (e) Different k: anti, $d = 5$, $n = 50K$. (f) Zillow data. $d = 4$, $n = 2M$.

Figure 4. Experimental results.

V. EXPERIMENTS

This section provides our benchmarks on synthetic and real data to reveal the performance of the outlined algorithms. 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) [15]. EBNL is a variant of the standard BNL algorithm [1] with the modification that each computed stratum is removed from the dataset and the Skyline is computed again. ESFS is an extension of SFS [5] exploiting some kind of data pre-sorting. In the worst-case EBNL and ESFS have a time complexity of $\mathcal{O}(n^2)$. The algorithms have been implemented in Java 7.0. TkLS follows the implementation details given in [9] and [11]. The experiments were performed on a machine running Debian Linux 7.1 equipped with an Intel Xeon 2.53 GHz processor.

For our synthetic datasets we used the data generator commonly used in Skyline research [1]. We generated *anti-correlated* (anti), *correlated* (corr), and *independent* (ind) distributions and varied (1) the data cardinality n, and (2) the data dimensionality d. For the experiments on real-world data, we used the well-known *Zillow* dataset from *www.zillow.com*. This dataset 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 Zillow dataset also serves as a real-world application which requires finding the Skyline on data with a low-cardinality domain.

Figures 4a – 4c present 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.

Figure 4d shows the runtime of all algorithms on 3, 5, and 7 dimensions having anti-correlated 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* elements. We also present the size of the different multi-level Skylines. For example, 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 4e visualizes the effect of different values of k. Therefore we computed top-k elements for $k \in \{100, 500, 1K, 25K\}$ using 5 dimensions (as in Figures 4a – 4c) and a data cardinality of $n = 50000$. The runtime for TkLS for all ks 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. 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.

Figure 4f presents our results on real data, i.e., the Zillow dataset (domain $[10; 10; 36; 46]$). Again, 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.

VI. Related Work

The idea of *multi-level Skylines* was already mentioned by Chomicki [3] under the name of *iterated preferences*. However, Chomicki has never presented an algorithm for their computation. Apart from that there is no other work on computing the i-th stratum of a Skyline query.

Regarding top-k [13] and Skyline [2] queries, there are some approaches that combine these both paradigms to *top-k Skyline queries*. In [14] and [15] 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. The authors of [16] abstract Skyline ranking as a dynamic search over Skyline subspaces guided by user-specific preferences. In [17][18] and [19] 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. [20] considers top-k combinatorial Skyline queries, and Zhang et al. [21] discuss a probabilistic top-k Skyline operator over uncertain data. Top-k queries are also of interest in the computation of spatial preferences [22], where the aim is to retrieve the k best objects in a spatial neighborhood of a feature object. Yu et al. [23] consider the problem of processing a large number of continuous top-k queries, each with its own preference. Although there is some related work, the problem of efficiently evaluating top-k Skylines is still an open issue.

VII. Conclusion and Future Work

We have found a promising algorithm for the iterated evaluation of a Skyline query. After a single run 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. Although *multi-level Skyline sets* have been introduced some years ago, no algorithms specialized on this problem have been proposed. Furthermore, our TkLS algorithm also provides an efficient way to compute top-k Skylines. The only restriction we suffer from are low-cardinality domains. Since this is a huge restriction to Skyline computation, we want to adjust our algorithms as suggested in [24]. They use some kind of down-scaling of the domain. Furthermore, on the basis of [9] we want to develop a parallel algorithm for top-k Skyline computation. But this remains for future work.

References

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

[2] J. Chomicki, P. Ciaccia, and N. Meneghetti, "Skyline Queries, Front and Back," SIGMOD, vol. 42, no. 3, 2013, pp. 6–18.

[3] 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.

[4] 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.

[5] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with Presorting," in Proceedings of ICDE '03, 2003, pp. 717–816.

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

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

[8] S. Liknes, A. Vlachou, C. Doulkeridis, and K. Nørvåg, "APSkyline: Improved Skyline Computation for Multicore Architectures," in Proceedings of DASFAA '14, 2014, pp. 312–326.

[9] M. Endres and W. Kießling, "High Parallel Skyline Computation over Low-Cardinality Domains," in Proceedings of ADBIS '14. Springer, 2014, pp. 97–111.

[10] M. Morse, J. M. Patel, and H. V. Jagadish, "Efficient Skyline Computation over Low-Cardinality Domains," in Proceedings of VLDB '07, 2007, pp. 267–278.

[11] T. Preisinger and W. Kießling, "The Hexagon Algorithm for Evaluating Pareto Preference Queries," in Proceedings of MPref '07, 2007.

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

[13] 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.

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

[15] 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, ser. LNCS, vol. 4653. Springer, 2007, pp. 254–263.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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. ACM, 2012, pp. 397–408.

[24] M. Endres, P. Roocks, and W. Kießling, "Scalagon: An Efficient Skyline Algorithm for all Seasons," in DASFAA '15: Proceedings of the 20th international conference on Database systems for advanced applications, 2015, in press.

 ISBN: 978-1-61208-408-4