Dynamic Skyline Computation with the Skyline Breaker Algorithm

Dominik Köppl

11. August 2014

Abstract

Given a sequential data input, we tackle parallel dynamic skyline computation of the read data by means of a spatial tree structure for indexing fine-grained feature vectors. For this purpose, we modified the Skyline Breaker algorithm that solves skyline computation with multiple local split decision trees concurrently. With this approach, we propose an algorithm for dynamic skyline computation that inherits the robustness against the "dimension curse" and different data distributions.

1 Introduction

Let us assume a group of tourists going sight-seeing in the historic downtown of Wroclaw. During their sight-seeing trip they want to locate a restaurant by querying a database with their mobile phone. Their current location is at a noisy construction site in an actual quite nice place. Hence, they prefer a restaurant that should be close to their current location, but not in direct vicinity of it. Moreover, they search an affordable dinner with exception of fast-food. But unfortunately, the database just classifies the locations by price. In order to cope with this restriction, they take a certain price as ideal. Alas, there might be no restaurant that matches both properties "affordable" and "nearby center", because fast-food stands and more expensive restaurants might also located closer to downtown. If the query result is unsatisfiable, the tourists might slightly modify their query ahead of the next try (e.g., by moving to another location or arguing about the price). Without an ingenious strategy, common skyline algorithms have to project the restaurants w.r.t. the query each time from scratch. Dynamic skyline algorithms provide fast results for recurring queries of the same kind by caching [20]. The key of dynamic skyline is a generalized mapping of the input data to the feature vector space. Instead of reconstructing this mapping for each query, it stays the same during recurring queries. Effectively, the vector space is parametrized by a query vector, i.e., it is shifted by an offset.

1.1 Dynamic Skylines

Consider we have a data set Ω and a scoring function $f : \Omega \to \mathbb{R}^n$. We call an element $a \in \Omega$ a tuple, and $f(a)$ its respective (feature) vector. Let us say that we have a query vector $q \in \mathbb{R}^n$ and want to compute the dynamic skyline of q [19], i.e., we want to find all vectors of $f(\Omega)$ that are not dominated by any other vectors w.r.t. q. A vector dominates another vector if its distance to q has an equal or lower value than the other vector's distance to q in all respective coordinates, and a strictly lower value in at least one coordinate. According to this definition, the query vector $q \in \mathbb{R}^n$ always dominates all other vectors, but q is not part of $f(\Omega)$ in general. We call q an offset vector, because it moves the best location $0 \in \mathbb{R}^n$ of the classical skyline problem to its own location (e.g. we translate the vector space by $\mathbb{R}^n \to \mathbb{R}^n, v \mapsto v + q$). Vectors are often illustrated as points of the $\mathbb{R}^n$ vector space equipped with the l_1 norm. The latter is useful, as two points are incomparable with respect to domination when their norm is equal.

1.2 Structure of the Paper

This paper provides a modification of the Skyline Breaker algorithm [16]. While the original paper suggests an algorithm for classic skyline computation, we extend its usage to dynamic skyline queries. Therefore, we review the main ideas before addressing our new enhancements. The rest of our paper is organized as follows: Section 2 contains an overview of related work with regards to skyline computation on tree data structures. We introduce basic definitions and helpful lemmata in Section 3 with the sLSD-tree structure in Section 4 before focusing on the algorithm itself in Section 5. There, we present the problem and highlight different techniques for optimizations. Finally, Section 6 reflects possible performance speedups and slowdowns.

2 Related Work

The approach of skyline-computation by means of geometric representation of data already cherishes a huge authorship. Kossmann et al. [15] propose nearest neighbor (NN) search operating on an R*-tree [1] structure. This idea was extended by Papadias et al. [18] featuring the branch-and-bound skyline (BBS) algorithm. They were also the first to propose the dynamic skyline problem [19]. Using the M-tree [7] as a geometrical representation for the data, Chen and Lian [6] reasoned about pruning techniques for fast dynamic skyline computation. Additional to providing a dynamic skyline algorithm, Sacharidis et al. [20] proposed caching methods to speed up computation in subsequent queries. Along with new proposals for skyline computation, the focus of recent research projects tends to parallelization of classic skyline algorithms. Selke et al. [21] discuss a variant of block-nested loop (BNL) using the Lazy List [9] data structure. Techniques like optimistic locking are shown as a good trade off between redundant synchronization steps and race conditions. By doing so, they manage to successfully shrink the sequential fraction of BNL. Next to BNL, a possible parallelization of BBS and SFS are treated by Im et al. [14]. Obviously, dynamic skylines and parallel skyline algorithms are part of an active research area. The combination of both fields is topic of this paper.

3 Preliminaries

As in common calculus, $\pi_i : \mathbb{R}^n \to \mathbb{R}$ denotes the projection to the i-th coordinate, i.e., for all $(v_1, \ldots, v_n) \in \mathbb{R}^n$

$$\pi_i(v_1, \ldots, v_n) = v_i \text{ with } i \in \{1, \ldots, n\}\,.$$

We call a function $f : \Omega \to \mathbb{R}^n$ a scoring function. Here Ω is a subset of an arbitrary universe, in an abstract sense. Further, for a scoring function $f : \Omega \to \mathbb{R}^n$, we call $v := f(x) \in \mathbb{R}^n$ the feature vector of $x \in \Omega$. We will now consider a fixed offset $q \in \mathbb{R}^n$. The motivation about the definitions is that f and q induce a strict weak ordering $\prec_q$ on Ω, called a dynamic scoring of Ω. For two vectors $v, w \in \mathbb{R}^n$, we call v better than w w.r.t. q if there exists a $j \in \{1, \ldots, n\}$ such that $|\pi_j(v - q)| < |\pi_j(w - q)|$ and $|\pi_i(v - q)| \le |\pi_i(w - q)|$ for every $i \in \{1, \ldots, n\}$. We shortly write $v \prec_q w$. Analogously, we write $x \prec_q y$ for two tuples $x, y \in \Omega$ if and only if $f(x) \prec_q f(y)$ holds.

In case there is no possibility of confusion, we write $|S|$ to express the cardinality of a set S, i.e., the number of elements of S.

3.1 Formal Problem Definition

The dynamic skyline parameterizes the classic skyline by a feature vector q as an offset [19]. More concretely, the dynamic skyline set $\mathbb{S}^q(\Omega) \subset \Omega$ holds the condition

$$\begin{aligned} x \in \mathbb{S}^q(\Omega) \quad :\Leftrightarrow \quad & \forall o \in \Omega, \text{ there either exists a } j \in \{1, \ldots, n\} \\ & \text{such that } |\pi_j(f(x) - q)| < |\pi_j(f(o) - q)| \\ & \text{or } f(x) = f(o) \\ \Leftrightarrow \quad & \text{there exists no } o \in \Omega \text{ with } o \prec_q x. \end{aligned}$$

We also call $x \in \mathbb{S}^q(\Omega)$ a tuple that is not dominated by any other tuple of Ω. In the following, when the sense is obvious, we identify a tuple $x \in \Omega$ with its value $f(x)$, e.g., we do not explicitly mention any notion of f. So when speaking about coordinates of x, we actually mean the coordinates of $f(x)$.

Lemma 1. *Let $U_1, \ldots, U_k \subset \Omega$ be a cover of Ω, i.e., $\bigcup_{i=1}^k U_i = \Omega$. Then $\mathbb{S}^q(\Omega) \subset \bigcup_{i=1}^k \mathbb{S}^q(U_i)$*

Proof. Let $x \in \mathbb{S}^q(\Omega) \cap U_i$, then for all $o \in U_i \subset \Omega$, there either exists a $j \in \{1, \ldots, n\}$ such that $|\pi_j(f(x) - q)| < |\pi_j(f(o) - q)|$ or $f(x) = f(o)$. Hence $x \in \mathbb{S}^q(U_i)$. $\square$

4 The sLSD-Tree

The LSD-tree is a hybrid tree that has bucket nodes as leaves and directory nodes as internal nodes. Moreover, it is a binary tree as a bucket split divides two bucket nodes with a hyperplane. Bucket nodes are the actual nodes which store the data. We call the data of this kind of node a bucket. On the other hand, directory nodes only save the split-position and the two bucket nodes split by the hyperplane. In the beginning, the tree possesses a single bucket node. Whenever a bucket contains more than M elements, this bucket node will be split. In our case, the split point is determined by the median of all elements according to one dimension. After the split, we have a left and a right bucket node: The left bucket contains those elements which have smaller values than the split-position in the split-dimension. The information of the split is saved in a new directory node that replaces the previous bucket node and adopts both new bucket nodes as children. Exactly as the kd-tree, the split-dimension can be determined by the tree-depth of the bucket node which shall be split. In another perspective, the LSD-tree partitions data of $\mathbb{R}^n$ into disjoint cuboids where each cuboid contains all elements of exactly one bucket. So each bucket can be represented as a cuboid which coordinates are saved in the ancestor nodes' split positions. We call this particular cuboid the bounding cuboid of the bucket. Computing the NN of an arbitrary point $q \in \mathbb{R}^n$ is done by the following steps: First traverse the tree in a top-down manner in order to locate the bucket, in which q would belong. Next, walk recursively upwards while analyzing at each depth the sibling for near points. Note that the sibling can be a subtree, hence we have to analyze it recursively, too. Henrich [10] proposes an algorithm that takes the distances to already found points for a bound to stop the naïve search prematurely. The difference between the LSD-tree and our sLSD-tree is the lack of external directory nodes as we are constraining the problem for in-memory use cases. Finally, our `sLSD` saves data of type Ω, but uses the function f to determine the position of the data.

4.1 Dealing with skewed distributions

The sLSD-tree splits an overfull bucket by sorting its elements in a certain dimension and taking the median. This dimension is determined by its parent node. In the worst case scenario, all elements share the same value in this dimension. A split would therefore create two new buckets; one assigned with the entire content and the other left empty. Henrich [12] avoids this bad behavior by simply iterating over the split dimension until he finds different values of the elements in this dimension. For termination, we just have to check if at least two elements have different feature vectors. Another method would take those dimensions as split dimension into account, for which only few objects' value collide with the median value in this dimension. If there are multiple dimensions with the same number of minimal collisions, then we take the dimension with the highest variance in objects' value out of this remaining set. This additional filter tries to hinder any bucket of becoming too elongated in specific dimensions.

4.2 Complexity Analysis

As a member of the kd-tree family, the sLSD-tree possibly shares complexity traits with its ancestor. In fact, the kd-tree is a specialization of the sLSD-tree with a maximum bucket size of $M = 1$. On the one hand, for neglectably small M, the sLSD-tree provides operations like inserting or locating an element in $\mathcal{O}(\log_2 |\Omega|)$ average and $\mathcal{O}(|\Omega|)$ worst time, too. On the other hand, a $M \geq |\Omega|$ lets the sLSD-tree store the entire data in a single bucket. Hence, the time complexity is based on the data structure used for the buckets. For the other cases with $1 \ll M < |\Omega|$, we have to take care for the more complex insertion step. If we assume contrarily to Subsection 4.1 that the median value is always unique under all considered objects during a split, then any split of an overfull bucket at the median creates two buckets, each containing at least $\lfloor M/2 \rfloor$ elements. Moreover, our algorithm will not remove any element of the sLSD-tree. After inserting $M + 1$ elements, there is at no time any bucket with less than $\lfloor M/2 \rfloor$ elements. Let us use the random variables $b_i \in [\lfloor M/2 \rfloor, M)$ with $i = 1, \ldots, k$ for counting the elements of each bucket of an sLSD-tree with k leaves. These variables satisfy the constraint that $\sum_{i=1}^{k} b_i = |\Omega|$. In the average case, the occupancy rate is distributed uniformly over all intervals, hence we get $\frac{3}{4}kM = |\Omega|$, i.e., $k = \frac{4|\Omega|}{3M} = \mathcal{O}\left(\frac{|\Omega|}{M}\right)$. Therefore, the sLSD-tree has an average depth of $\mathcal{O}\left(\log_2 \frac{|\Omega|}{M}\right)$. The costs for inserting an element are a combination of

a) finding the bucket in which the element shall be inserted. The time is logarithmic in the depth of the tree.

b) splitting this bucket if it is overfull. The split is done by sorting the elements in one dimension that takes $\mathcal{O}(M \log_2 M)$ time and creating a new bucket that takes $\mathcal{O}(M)$ time for moving half of the elements to the new bucket. Overall, a newly created bucket can take $M/2$ new elements in the average before it has to be split. Hence, this step takes $\mathcal{O}(\log_2 M)$ amortized time.

To sum up, insertion takes $\mathcal{O}(\log_2 |\Omega|)$ amortized time in average. In the worst case, all buckets have minimal occupancy, i.e., $b_i = \lfloor M/2 \rfloor$. Thus, we have $k = \frac{2|\Omega|}{M}$. Let us take an empty sLSD tree. If we insert a sequence of elements Ω that are strictly ordered, we obtain, like for kd-trees, a caterpillar tree with the worst case depth of $\mathcal{O}(\Omega)$ if we drop the restraint of unique values. It is easy to see that the distribution of elements will not only affect the number of buckets, but also the time complexity. Fortunately, Henrich [11] provides a heuristic online-strategy that tries to redistribute the occupancy. His strategy tries to shift the split position of the parent node of an overflowing bucket in order to prevent the split of the bucket. But we have to take into account, what kind of node the sibling of the brimful bucket is:

a) If it is a bucket that has space left, we just shift the parent node's split position such that the number of elements gets rebalanced.

b) If the sibling is a subtree that can adopt a new element without splitting one of its children, we shift again the split position of the bucket's parent node. But this time, we also have to recompute the split information of the interim nodes of this subtree.

If none of these conditions can be applied, we declare the bucket's parent node as overfull and try to shift the split position of its respective parent. Hence, if a local redistribution is not possible, we recursively go upwards the tree and reconstruct the split information. Regarding aggressive shifting strategies, there is a certain trade-off between overall bucket utilization and the number of directory nodes [11].

4.3 Geometrical Characteristics

In the context of the Skyline Breaker algorithm we are particularly interested in catching skyline points out of the sLSD-tree quickly. Therefore, we need to compute the bounding cuboids for nodes with arbitrary depth. For that, we denote with $\operatorname{depth}(e)$ the depth of any node e of an sLSD-tree. The following lemmas will help us to solve this task.

Lemma 2 ([16])**.** *Let b be an arbitrary leaf node. To determine the bounding cuboid of b we need to examine n ancestors of b.*

Let $o = (o_1, \ldots, o_n) \in f(\Omega)$ be a feature vector and $q = (q_1, \ldots, q_n)$ the query vector.

Lemma 3. *The pairwise disjoint, non-bounded cuboids $D_1 := (q_1 + d_1, \infty) \times (q_2 + d_2, \infty) \cdots \times (q_n + d_n, \infty), D_2 := (-\infty, q_1 - d_1) \times (q_2 + d_2, \infty) \cdots \times (q_n + d_n, \infty), \ldots, D_{2^n} := (-\infty, q_1 - d_1) \times \cdots \times (-\infty, q_n - d_n))$ do not contain any skyline point, where $d := (|q_1 - o_1|, \ldots, |q_n - o_n|)$. Furthermore, $\mathbb{R}^n \setminus \bigcup_{j=1}^{2^n} D_j$ is connected.*

Proof. For every feature vector $u \in \bigcup_{j=1}^{2^n} D_j$ we have $|\pi_j(u - q)| > |\pi_j(o - q)|$ for every $j = 1, \ldots, n$, and thus u is dominated by o. $\square$

Lemma 4. *Both the siblings of B_q and the descendants of B_q's ancestors with depth at least $\operatorname{depth}(B_q) - n + 1$ can contain skyline points.*

Proof. W.l.o.g. we assume $\operatorname{depth}(B_q) > n - 1$. Let p be the ancestor of B_q with a depth of $\operatorname{depth}(B_q) - n$ and let c be the child that is the ancestor of B_q. The node c has a depth of $\operatorname{depth}(B_q) - n + 1$. The descendant directory nodes of c and c itself cannot have any split position farther away from q than p. Otherwise, the bucket B_q would be at a different location than q. Because of the definition of B_q, no directory node that is both ancestor of B_q and descendant of p has the split-dimension of p. Hence, there is no leave node of c whose bounding cuboid is contained (entirely) in D (of Lemma 3 with $B_o \leftarrow B_q$). $\square$

Remark 1. *According to Lemma 4, if the depth of the sLSD tree is less than $n - 1$, the pruning technique cannot be done.*

5 The Algorithm

While the original algorithm restricts itself to non-negative feature vectors, we had to relinquish this constraint in order to parametrize the skyline computation with an arbitrary offset q. Therefore, we reintroduce basic original concepts, while providing our alternations.

5.1 The Dynamic Skyline Breaker Algorithm

We generate an `sLSD`-tree for each thread. Each thread starts with the `SBDyn` (Algorithm 1) immediately after its tree is filled with the input data. Firstly, we search the nearest neighbor of q with respect to the l_1 norm in terms of bucket nodes by Henrich's distance-scan algorithm [12]. While locating this node, at most $2^n - 1$ buckets have to be examined. Because we have not yet found any dominator, these buckets might contain skyline points additional to a. Hence, we collect these buckets for local skyline computation and call them $\{B_i\}_{i \in I}$ for some I. Let $a \in f(\Omega) \subset \mathbb{R}^n$ be a NN of q and let B_q denote the bucket in which q would be added. Now, having Lemma 2 in mind, we start to traverse the tree reversely, counting our steps upwards. We initialize a vector $v \leftarrow q$ that keeps track of the geometrical position. While climbing the tree upwards, we examine the sibling c of the node we came from. Until we have not counted upwards to the dimension n, at least one of v's coordinates is zero (see Lemma 4). So we cannot discard c. In other words, we need to traverse c and collect all of its buckets. When we have reached the number n with our counting, one of the descendants of c could be discarded. That is because some descendants' bucket might be contained in the region without skyline points considered in Lemma 3. Therefore, we start a new task that traverses c, done by the tree clipping, described in Section 5.3 and Algorithm 2. We have traversed the tree when we have reached the root node while climbing up the tree from B_q. The last step is the allocation of the global skyline.

5.2 Skyline in a Bucket

For local skyline calculation we can modify any skyline algorithm to work with an offset of q. We did this with a simple BNL-based algorithm that works directly on the bucket, while subtracting the offset ahead of comparison.

5.3 The Tree Clip Algorithm

Algorithm 2 works with a divide and conquer (DC) strategy. We start with a directory node and the split information of its parent node. It is advisable to hold this data in a variable $v \in \mathbb{R}^n$ by setting the coordinate entry of p at the split-dimension to the split-position, while all other coordinates are kept identical to q's position for a start. The other child c of our current directory node can be illustrated as a cuboid with the corner point $e_c = v$ that is closest to q. So, while we traverse the children of our current directory node, we

Algorithm 1 Dynamic Skyline Breaker

```
function $\textsc{sbDyn}(q \in \mathbb{R}^n)$
    $Q \leftarrow \text{Set}(\emptyset) \subset \Omega$
    $(a, B_q, \{B_i\}_i) \leftarrow \textsc{NearestNeighbor}(q)$
    $\textsc{Assert}(a = \text{minarg}_{a \in f(\Omega)} \|a - q\|_{l_1})$
    $Q.\texttt{insert}\left(\mathbb{S}^q\left(B_q \cup \{B_i\}_i\right)\right)$
    $N \leftarrow B_q$
    for $i \leftarrow 0$ to $n$ do
        $c \leftarrow N, N \leftarrow N.\texttt{parent}$
        $c \leftarrow N.\texttt{left} = c\ ?N.\texttt{right} : N.\texttt{left}$
        $Q.\texttt{insert}\left(\mathbb{S}^q\left(c\right)\right)$
    end for
    while $N \neq root$ do
        $c \leftarrow N, N \leftarrow N.\texttt{parent}$
        $c \leftarrow N.\texttt{left} = c\ ?N.\texttt{right} : N.\texttt{left}$
        if $c$ is a BucketNode then
            $Q.\texttt{insert}\left(\mathbb{S}^q\left(c\right)\right)$
        else
            $v \leftarrow q \in \mathbb{R}^n$
            $w \leftarrow$ split-position of $N$
            $v.$(split-dimension of $N$) $\leftarrow w$
            $\textsc{clip}(q, c, v, Q)$
        end if
    end while
    return $\mathbb{S}^q\left(Q\right)$
end function
```

note down the split information in v until we have gained coordinates for all dimensions. With this information represented by the split point $v \in \mathbb{R}^n$, using the condition $a \prec_q v$, we now check whether discarding the child node c is feasible. If the condition holds, we know that all elements contained in the cuboid of c are dominated by a. Thus, we discard the node c, even if it is a directory node. Otherwise, we have to traverse the right node recursively, again denoting its split data.

6 Analysis and Optimization

Let us retrace the versatile steps involved in the algorithm while analyzing the time complexity. At the start, locating B_q will take $\mathcal{O}\left(\log_2 \frac{|\Omega|}{M}\right)$ in the average and $\mathcal{O}\left(|\Omega|\right)$ in the worst case, according to Subsection 4.2. Subsequently, a is found while examining at most $2^n - 1$ that takes $\mathcal{O}\left((2^n - 1)M\right)$ time in the worst case. If we take a balanced sLSD-tree for granted, the preparation step of Subsection 5.3, that moves upwards until reach-

Algorithm 2 Clipping the Tree

```
function CLIP($q \in \mathbb{R}^n, N : \text{node}, v \in \mathbb{R}^n, Q \subset \Omega$)
    $w \leftarrow$ split-position of $N$
    $d \leftarrow$ split-dimension of $N$
    $c \leftarrow q.d < w\ ?N.$left : $N.$right
    if $c$ is a BucketNode then
        $Q.$insert $(\mathbb{S}^q(c))$
    else
        clip($q, c, v, Q$)
    end if
    $v' \leftarrow v$
    $v'.d \leftarrow w$
    if $a \not\prec_q v'$ then
        $c \leftarrow c = N.$left $?N.$right : $N.$left
        if $c$ is a BucketNode then
            $Q.$insert $(\mathbb{S}^q(c))$
        else
            clip($q, c, v', Q$)
        end if
    end if
end function
```

ing at least n different dimensions, will also take $\mathcal{O}(2^n)$ buckets into consideration. For local skyline computation of any bucket, an algorithm like BNL [5] applied in Subsection 5.2 needs at least $\Omega(M)$ time and has $\mathcal{O}(M^2)$ worst time complexity. The time taken by the actual tree clipping algorithm is heavily dependent of the distribution of split information and takes at least time proportional to the remaining depth of the current node in the sLSD-tree after the preparation step. By building our algorithm upon the sLSD-tree we yield a combinatorial approach that employs both the DC-strategy of Subsection 5.3 and some nested-loop algorithm of Subsection 5.2. Both strategies are weighted by the maximum bucket size M. On the one extreme, for $M = 1$ we get a kd-tree like structure with singletons as buckets. According to the complexities of Subsection 4.2, setting $M = 1$ creates a much larger tree structure and exploits entirely the DC-strategy. On the other hand, taking any $M > |\Omega|$ puts the entire content in a single bucket and just evaluates the nested-loop algorithm. For taking advantage of both algorithms, a $M \in (1, \frac{|\Omega|}{2^{n+1}}]$ has to be chosen, dependent of the input data's distribution. Obviously, there are data sets for which choosing a small or large M results in a speed-up. A best-case scenario for large M is a correlated data set with $\mathcal{O}(|U|)$ time complexity for the BNL of any subset $U \subset \Omega$. If the sLSD-tree is balanced, we have a depth of $\mathcal{O}\left(\log_2 \frac{|\Omega|}{M}\right)$ and thus $\frac{|\Omega|}{M}$ leaves. Hence, the algorithm will take at most $\mathcal{O}(|\Omega| \log_2 |\Omega|)$ time. On the other hand, extremely anti-correlated data can pose the worst case running time for the BNL. But if the clipping condition holds for most of the time, we have for $\frac{|\Omega|}{M} \geq 2^n$ the best case running time $\mathcal{O}\left(\log_2 \frac{|\Omega|}{M} - n\right)$ for the tree climbing algorithm that collects $\mathcal{O}\left(2^n\right)$ buckets of a balanced LSD-tree in addition. For merging these buckets, we have $2^n\mathcal{O}\left(M\right)$ best case and $2^{2^n}\mathcal{O}\left(M^2\right)$ worst case time. Note that both times are independent of $|\Omega|$. Hence, we argue that the dimensional factor plays only a minority role for reasonable large $|\Omega| > 2^{2n}$.

Proposition 1. *The Skyline Breaker algorithm accesses the optimal number of nodes for computing the dynamic skyline set of q, i.e., no algorithm will solve the same problem while visiting fewer nodes on the same data structure without knowledge of the contents of any bucket.*

Proof. Let us assume that our algorithm unnecessarily inspects a bucket u that can be safely pruned. The bucket u has a bounding rectangle with corner points $(u_0, \ldots, u_{2^n})$. Let u_L be a best corner, i.e., $u_L \prec_q u_i$ for every $i = 1, \ldots, 2^n$. In the contrary, there has to be an element $v \in \Omega$ with $f(v) \prec u_L$. Otherwise, without further knowledge, we have to inspect the bucket u. We also know that the NN of q is always part of the skyline set. Hence, every algorithm that computes the dynamic skyline set has to access the bucket B_a that would take a as a new element of the global skyline. Due to the fact that Algorithm 2 traverses the tree from B_q upwards, we will encounter nodes in an ascendingly, weakly sorted order w.r.t. its best corner. In other words, the algorithm will either find a bucket containing v before accessing u or find a bucket that dominates v. In both cases, we will not inspect the bucket u; either we clip a subtree containing u or u itself. $\square$

6.1 Improving the Clipping Condition

The tree clipping (Algorithm 2) could be extended to use a list of known skyline points instead of solely a (a as defined in Section 5.1). A large list $\mathbb{S}' \subset \mathbb{S}$ of global skyline points reinforces the clipping in Subsection 5.3 by testing $o \prec_q p$ for any $o \in \mathbb{S}'$ instead of only a. Thus a skyline subset will provide a higher chance of clipping. Actually, we can already enlist some points while finding a; let us recall that the search for the NN of q can take at most $2^n - 1$ buckets $\{B_i\}_i$ into account. Due to the distance-scan algorithm [12], each of these buckets is a neighbor of B_q. The possibly huge number arises from the number of orthants that are spawned by q (see Lemma 3). Because the bounding cuboids form a connected region that contains q, there exists a certain area in which we can be sure that local skyline points belong to the global dynamic skyline. More precisely, we compute the maximal cube (i.e., a cuboid with equal sides) that is contained in the union of all buckets. We denote its side length with s. Then any $o \in \mathbb{S}^q\left(\bigcup_i B_i\right)$ with $\|o - q\|_{l_1} \leq s$ is in the global skyline $\mathbb{S}^q\left(\Omega\right)$. If there would be a $p \prec_q o$ then $\|p - q\|_{l_1} \leq s$ and hence $p \in \mathbb{S}^q\left(\bigcup_i B_i\right)$, a contradiction for o being a local skyline point. By taking any yet encountered local skyline point for filtering during the clipping, Proposition 1 is tightened to the statement that the algorithm is optimal w.r.t. the class of algorithms that may use knowledge of the buckets' contents, too.

6.2 Analysis of Parallelism

At first glance, it seems an easy task to divide the algorithm in independent subtasks: Firstly, the input data is partitioned and each part gets processed by a different thread. Particularly for a concurrent input source like in RAM stored data of a CRCW, there is no sequential bottleneck. In the main step, each thread computes independently the skyline of its fetched input data. Finally, we just merge the local skylines until a single (i.e., the global) skyline remains. Unfortunately, the described algorithm is not embarrassingly parallel; it has several delicate parts for that we have to take care when adding concurrent structures to our code. In particular, this involves the queue that holds the computed, local skylines. For this job, Java 7's `ConcurrentLinkedQueue` with wait-free access support [17] seemed fitting for us. Moreover, before climbing its tree, every thread must have a NN of q in order to exploit the clipping technique. This may be either a point of the thread's own input data set, a global NN of q or even a list comprising every already found skyline point. Hence, we either use a global barrier or a concurrent list that is filled with newly found local skyline points and trimmed when an already stored point gets dominated by a recently collected point; this point cannot be dominated by another point of the list, otherwise the trimmed point would have been trimmed in advance. The other parts of the implementation are rewritten in the MapReduce and Fork/Join model. Both are two different models for parallel execution. They share the common goal to ease parallelization of a given sequential task. Nevertheless, their field of application is orthogonal. While MapReduce targets distributed computing, Fork/Join works merely on a single node. We will describe our algorithm as a hybrid approach that employs both models — MapReduce for distributing the work-load to different nodes and Fork/Join for local multi threaded computation. For the latter, we just translate the ideas of [16] to dynamic skyline computation. In order to describe our idea in the MapReduce framework, let us take a partition $\{U_j\}_{j \in J}$ of Ω, i.e., $\bigcup_{j \in J} U_j = \Omega$ with $U_i \cap U_j = \emptyset$ for $i, j \in J$ pairwise different. The three functions $\mathtt{map} : (\{j\}, U_j) \mapsto (\{j\}, \mathbb{S}^q(U_j))$, $\mathtt{combine} : ((a, \mathbb{S}^q(A)), (b, \mathbb{S}^q(B))) \mapsto (a \cup b, \mathbb{S}^q(A \cup B))$ for each $a, b \subset J$ with any $A, B \subset \Omega$ and $\mathtt{reduce} : (J, \mathbb{S}^q(\Omega)) \mapsto \mathbb{S}^q(\Omega)$ represent the common work flow of our MapReduce application. In particular, our implementation is a deterministic, multi threaded program [3], because its schedule is predetermined by a fixed number of tasks that consists of the starting threads and the threads that combine the local skylines. Nevertheless, the code of Subsection 5.3 is non-deterministic w.r.t. thread spawning. That is caused by the fact that the number of tasks is dependent of the effectiveness of the clipping and the recursive divide and conquer technique. The latter tells us that the tree clipping is a fully strict computation [2]. Because common frameworks like Hadoop do not allow dynamic task creation [8], the clipping cannot be described in terms of the MapReduce model. Fortunately, translating this part of the algorithm to the Fork/Join model seemed the right choice; Blumofe et al. showed that the work-stealing technique is optimal for scheduling fully-strict computations.

6.3 Caching Dynamic Skylines

The construction of the sLSD-trees in Subsection 5.1 is done on the fly, i.e., the tree is not getting rebalanced during the bulk insertion. For reuse, we can restructure the sLSD-trees to prevent worst case scenarios for next queries. Therefore, we exchange our median-based split strategy with a distribution dependent one [13] in order to cope, for instance, with skew distributed data. An optimal refactoring of the tree results in a balanced tree with $\mathcal{O}\left(\frac{|\Omega|}{M}\right)$ depth. Additionally, we save after each query the offset vector q along with its dynamic skyline set $\mathbb{S}^q(\Omega)$ for reuse [20]. For effective caching we have to introduce the notion of orthants of q:

Definition 1. *The offset q splits the space into 2^n orthants. If we give each orthant a number, we can write the j-th orthant of q as O_j^q for every $j = 1, \dots, 2^n$. By Lemma 1, we have $\mathbb{S}^q(\Omega) \subset \bigcup_{j=1}^{2^n} \mathbb{S}^q\left(O_j^q\right)$.*

Now, Proposition 1 and a variation of [20, Lemma 2] comes in handy:

Lemma 5. *Let q' be an old offset and b a node of the sLSD-tree whose bounding cuboid $C_b \subset O_j^{q'}$ belongs to the j-th orthant of q', but does not contain any points of the j-th orthant skyline set $\mathbb{S}^{q'}(O_j^{q'})$. If q is an offset and q' belongs to the j-th orthant of q, then b does not intersect with $\mathbb{S}^q(\Omega)$ and thus can be clipped.*

The cache is used to provide additional conditions for the clipping algorithm. But before employing the cache, we take only these elements into consideration that effectively help clipping:

Corollary 1 ([20]). *Let q' and q'' be two old queries for that we cached their results. If $q' \prec_q q''$, then elements of the dynamic skyline-set of q'' are either part or dominated by elements of $\mathbb{S}^{q'}(\Omega)$. Hence, q'' will not clip additional nodes of the sLSD-tree.*

7 Conclusion

We have been taken on the shoulders of the Skyline Breaker algorithm that grants us the ability to handle high-dimensional features regardless of the input data's distribution. Some parts of the parallel algorithm were changed in order to cope with the dynamic version of skyline computation. The theoretical evaluation addresses the combinatorial nature of the algorithm and treats both best and worst case scenarios w.r.t. different maximum bucket sizes.

Acknowledgement

We are grateful to Dominik Kopczynski and Tomohiro I for proofreading this paper.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD Conference*, page 322–331. ACM Press, 1990.

[2] R. D. Blumofe. Executing Multithreaded Programs Efficiently. Technical report, 1995.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. PPOPP '95, page 207–216. ACM, 1995.

[5] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proceedings of the 17th International Conference on Data Engineering*, page 421–430. IEEE Computer Society, 2001.

[6] L. Chen and X. Lian. Dynamic Skyline Queries in Metric Spaces. EDBT '08, page 333–343. ACM, 2008.

[7] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. VLDB '97, page 426–435. Morgan Kaufmann Publishers Inc., 1997.

[8] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[9] S. Heller, M. Herlihy, V. Luchangco, M. Moir, S. William, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007.

[10] A. Henrich. A Distance Scan Algorithm for Spatial Access Structures. In *ACM-GIS*, page 136–143, 1994.

[11] A. Henrich. Improving the Performance of Multi-Dimensional Access Structures Based on k-d-Trees. In *ICDE*, page 68–75. IEEE Computer Society, 1996.

[12] A. Henrich. The LSDh-Tree: An Access Structure for Feature Vectors. In *ICDE*, page 362–369. IEEE Computer Society, 1998.

[13] A. Henrich, H.-W. Six, and P. Widmayer. The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects. In *VLDB*, page 45–53. Morgan Kaufmann, 1989.

[14] H. Im, J. Park, and S. Park. Parallel skyline computation on multicore architectures. *Inf. Syst.*, 36(4):808–823, 2011.

[15] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*, page 275–286. Morgan Kaufmann, 2002.

[16] D. Köppl. Breaking Skyline Computation down to the Metal - the Skyline Breaker Algorithm. In *IDEAS*. ACM, 2013.

[17] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. PODC '96, page 267–275. ACM, 1996.

[18] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *SIGMOD Conference*, page 467–478. ACM, 2003.

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

[20] D. Sacharidis, P. Bouros, and T. Sellis. Caching Dynamic Skyline Queries. In *Scientific and Statistical Database Management*, volume 5069, page 455–472. 2008.

[21] J. Selke, C. Lofi, and W.-T. Balke. Highly Scalable Multiprocessing Algorithms for Preference-Based Database Retrieval. In *Database Systems for Advanced Applications*. Springer, 2010.