Breaking Skyline Computation down to the Metal - the Skyline Breaker Algorithm

Dominik Köppl
Department of Computer Science, University of Augsburg, Germany
dominik.koeppl@informatik.uni-augsburg.de

ABSTRACT

Given a sequential input connection, we tackle parallel skyline computation of the read data by means of a spatial tree structure for indexing fine-grained feature vectors. For this purpose, multiple local split decision trees are simultaneously filled before the actual computation starts. We exploit the special tree structure to clip parts of the tree without depth-first search. The split of the data allows us to do this step in a divide and conquer manner. With this schedule we seek to provide an algorithm robust against the "dimension curse" and different data distributions.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Selection process; D.1.3 [**Programming Techniques**]: Parallel programming

Keywords

Skyline computation, concurrent computing, preference evaluation

1. INTRODUCTION

1.1 An Example of Skyline Search

Let us assume a group of tourists in the historic downtown of Barcelona. During their sight-seeing trip they want to locate a restaurant by querying a database with their mobile phone. They prefer a restaurant that should be affordable and nearby their current location. Unfortunately both properties "affordable" and "nearby center" can be anti-correlated. This means that the more expensive restaurants are also located closer to downtown. A common traditional database has no idea how to evaluate this selection and either provides an empty result set or shows the complete data set. The tourists are actually interested in those restaurants which are neither further away nor more expensive than any other restaurant. A set of these restaurants is called the skyline set [5]. By providing the skyline set as the result set of a database query, the tourists have an optimally pre-selected list for choosing their preferred restaurant.

IDEAS '13 October 09 - 11 2013, Barcelona, Spain

ACM 978-1-4503-2025-2/13/10 ...$15.00
http://dx.doi.org/10.1145/2513591.2513637 .

1.2 The Skyline Set

Consider we have a sequential input stream of a data set Ω and a scoring function $f : \Omega \to \mathbb{R}^n_{0,+}$. $\mathbb{R}^n_{0,+}$ is the space $\mathbb{R}^n$ without vectors with negative entries in any coordinate. We call an element $a \in \Omega$ a tuple, and $f(a)$ its respective (feature) vector. In case there is no possibility of confusion, we write $|A|$ to express the cardinality of a set A, i.e. the number of elements of A. We want to solve the maximal vector problem of the set $f(\Omega)$. The maximum vector problem [15][9] involves searching for all vectors of $f(\Omega)$ that are not dominated by any other vectors. A vector dominates another vector if it has an equal or lower value than the other vector in all respective coordinates, and a strictly lower value in at least one coordinate. For example, $0 \in \mathbb{R}^n_{0,+}$ always dominates all other vectors, but zero is not part of $f(\Omega)$ in general. The set of all vectors that are not dominated by others is called the Pareto set. 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. In our case, the l_1 norm of $f(\Omega) \subset \mathbb{R}^n_{0,+}$ can be simplified to a simple sum, i.e. $\|v\|_{l_1} = \sum_{i=1}^{n} v_i$ for $v = (v_1, \ldots, v_n) \in f(\Omega)$. Analogously, a tuple is part of the so called skyline set, if it is mapped to an element of the Pareto set by f. Hence our problem depends on the input size $N := |\Omega|$, the dimensionality n of the codomain of f and the distribution of the tuples. The distribution of $f(\Omega)$ has an enormous influence on the speed of a skyline solver. If we consider anti-correlated data like in our introduction in Section 1.1, many restaurants are either very cheap, very near to the center or mediocre in price and distance. That is why we can expect a very broad skyline. On the other hand, correlated data will tend to yield a small skyline with (relatively) many dominations. A particularly huge N bears the risk of the main memory being unable to hold the data needed for computation. To oppose this threat, algorithms can feature externalization techniques to utilize external media to prevent a great loss of speed. As memory size has been growing exponentially over the past decades, in-memory databases are currently suppressing traditional database systems in high performance areas. For that rea-

son, we do not consider externalization. Instead, we focus on the trend towards larger data sets caused by the increasing availability of information and the support of current hardware architectures. The phenomenon of the growing number of data sets with huge magnitudes is also called "big data". With this phenomenon a new problem arises, as it gets more and more difficult to analyze and process the entire dataset. As a consequence, new techniques to speed up computation when working with big data have to be found.

1.3 Concurrency

The doubling of transistors in central processing units (CPU) every two years since approximately 1960, also called "Moore's Law", can still be observed in the recent development of chip manufacturers. The Law correlated with the clock rates of the CPU and thus also influenced the performance of algorithms run on a newer CPU. But during the past decade, the frequency had hit a physical obstacle. Instead of leveraging the clock rate, CPU manufactors began to add multiple cores on one chip resulting in multi-core processors. Unfortunately, sequential algorithms do not benefit from this trend [18]. By means of concurrent programming, algorithms can take advantage of multi-core platforms. Concurrency is the concept of spawning, maintaining and joining multiple threads, using communication and synchronization between certain threads. Concurrent programming can lead to a parallel execution of code when the number of threads does not exceed the hardware-specific limit and idling of threads can be prevented. A good description of potentially beneficial effects is delivered by Gustafson's Law [10], stating that the speedup of concurrently programmed algorithms correlates with larger input data. A key factor is the proportionality between parallel and sequential parts of the algorithm. Synchronizations and barriers ordering threads to wait can slow down the effects of concurrent programming.

1.4 Structure of the Paper

One can easily conclude that concurrent skyline computation gained great importance in the field of database research in the past years. Therefore, we like to add an interesting solution to the described problem. The rest of our paper is organized as follows: Section 2 contains an overview of related work with regards to skyline computation on a tree data structure. We introduce basic definitions and helpful lemmata in Section 3 before focusing on the algorithm itself in Section 4. There, we present the problem and highlight different techniques for optimizations. Based on this knowledge, Section 5 reflects possible performance speedups and slowdowns. In Section 6 we challenge our developed algorithms against Hexagon of Preisinger and Kießling [21], BNL of Börzsönyi et al. [5] and BNL++ of Preisinger et al. [22]. Finally, we summarize our results and give an outlook on future work in Sections 7 and 8 respectively.

2. RELATED WORK

The approach of skyline-computation by means of geometric representation of data already cherishes a huge authorship. Kossmann et al. [14] propose nearest neighbor (NN) search operating on an R*-tree [2] structure. This idea was extended by Papadias et al. [19] featuring the branch-and-bound skyline (BBS) algorithm. An approach very close to our idea is a divide and conquer (DC) strategy on 2-dimensional space presented by Lu et al. [17]. Their DC-Skyline algorithm first targets the lowest left rectangle of a given R-tree. According to this rectangle they divide the space around it into regions, analogously to [14]. This division is also quite similar to our considerations in Figure 2. For the next idea we imagine the two-dimensional R-tree structure as cascading rectangles around the leaf nodes. For each bunch of nodes we can find a bounding rectangle that is again represented as a node in the R-tree. Now it is easy to see that we can discard all nodes (along with its children) which rectangular representations are contained in the region dominated by the found rectangle. So the algorithm proceeds by discarding these and splits recursively up for local skyline search in those areas that can still contain skyline points. Each recursive call takes only those nodes into consideration for which its representation as a rectangle intersects with a given region. On the one hand, as our algorithm is based on the LSD-tree that partitions space into disjoint subsets, we do not have to look for intersections. But on the other hand, the R-tree has the advantage that every node of the tree stores the complete data of a rectangle that contains its leaves. Thus looking up an area while traversing the R-tree is cheaper because the LSD-tree's nodes only store their split-position in one dimension. Thus a lookup of this kind takes either prior knowledge or some backtracking into account. Another difference lies in the fact that only the two-dimensional case is treated in their paper.

Given that the R*-tree's performance declines rapidly when the space has more than five dimensions, the problem of high dimensionality received tremendous sensation in the last two decades. Stefan et al. [24] propose to enhance the R*-tree with supernodes yielding the X-tree as a hybrid tree structure. Whereas the R*-tree can be favored for 2-dimensional computation, the X-tree clearly outperforms the family of R-trees in spaces with more than three dimensions. Unfortunately, the X-tree also lacks performance for increasing dimensionality. Böhm et al. [4] and Gaede and Günther [8] have already dealt with the problem of dimensionality and different tree structures. Another approach to avoid this "curse of dimensionality" was done by Bentley [3] with local hyperplane-splitting yielding the kd-tree. Henrich et al. [12] followed this concept with the local split decision tree (LSD-tree). Their LSD-tree saves the actual data as leaves in buckets and thus provides a heterogeneous tree with directory and bucket nodes. Based on the LSD-tree, we propose a new algorithm for skyline computation that features LSD-tree's robustness against a variance of dimensionality. By deploying a simplified version of the LSD-tree as underlying structure, we can apply a DC-strategy in certain conditions.

Another prosperous area of research is the examination of subsets of $f(\Omega)$ to avoid struggling with high dimensionality. Chan et al. [6] innovated a new measure that favors those tuples whose feature vectors belong to the skyline set in a majority of subsets. Their algorithm based on this measure can successfully approximate top-k computation. In some scenarios [20] it suffices to leverage the problem of skyline computation by determining only the skylines of a set of subspaces of the codomain of f. Clearly, the union of skylines of subspaces is a subset of the global skyline. Yuan et al. [27] introduce the problem of calculating the union of skylines of all subspaces. For this problem, they propose a DC strategy that computes skylines of different subspaces simultaneously, but not independently, while sharing common

data to prevent repeated computation of same tasks.

Along with new proposals for skyline computation, the focus of recent research projects tends to parallelization of skyline algorithms. Selke et al. [23] discuss a variant of BNL using the Lazy List [11] 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. [13]. Their approach consists of collecting incomparable tuples as the first step until a certain threshold is reached. After this sequential preprocessing dominance tests can be applied in parallel. Following up the proposed parallelizations of known algorithms they introduce the SSkyline algorithm that is similar to Best [25], but without the focus on top-k retrieval. For a concurrenct version of SSkyline, the authors propose two different approaches: Their first idea consists in enhancing SSkyline with concurrency in certain steps. The second method makes use of a model similar to MapReduce that maps the sequential algorithm SSkyline in parallel and merges the computed skylines as final step. They called the former resulting algorithm Parallel Skyline, the latter one PSkyline. We use resembling paradigms as our `SBFork` algorithm is a version of `SBSingle` with concurrent control structures, similar to the concept of Parallel Skyline. Like PSkyline, `SBQuick` beforehand splits the problem into small problems that are going to be solved independently by each thread.

3. PRELIMINARIES

3.1 Basic Definitions

As in common calculus, $\pi_i : \mathbb{R}^n \to \mathbb{R}$ denotes the projection to the i-th coordinate, i.e.

$$\pi_i(v_1, \ldots, v_n) = v_i \quad \forall (v_1, \ldots, v_n) \in \mathbb{R}^n, i \in \{1, \ldots, n\}$$

Let us call the space $\mathbb{R}^n_{0,+} \subset \mathbb{R}^n$ the space of feature vectors defined by

$$v \in \mathbb{R}^n_{0,+} :\Leftrightarrow \forall j = 1, \ldots, n : \ \pi_j v \geq 0.$$

We call a function $f : \Omega \to \mathbb{R}^n$ a scoring function if and only if $f(\Omega) \subset \mathbb{R}^n_{0,+}$ holds. Here Ω is a subset of an arbitrary universe, in an abstract sense. Further, for a scoring function $f : \Omega \to \mathbb{R}^n_{0,+}$, we call $v := f(x) \in \mathbb{R}^n_{0,+}$ the feature vector of $x \in \Omega$. The motivation about the definitions is that f induces a strict partial order $\prec$ on Ω, called a scoring of Ω. For two vectors $v, w \in \mathbb{R}^n_{0,+}$, we call v better than w if there exists a $j \in \{1, \ldots, n\}$ such that $\pi_j v < \pi_j w$ and $\pi_i v \leq \pi_i w$ for all $i \in \{1, \ldots, n\}$. We shortly write $v \prec w$. Analogously, we write $x \prec y$ for two tuples $x, y \in \Omega$ if and only if $f(x) \prec f(y)$ holds.

3.2 Formal Problem Definition

We are interested in the determination of the skyline set. The skyline set $\mathbb{S} \subset \Omega$ is the set for which the condition

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

is valid. We also call $x \in \mathbb{S}$ 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 we speak about coordinates of x, we actually mean the coordinates of $f(x)$.

3.3 The sLSD-Tree

As our algorithm tries to locate the nearest neighbor (NN) of zero as the primary step, our focus lay in a data structure which is fast at searching for the NN of zero. It has been proven experimentally that calculating the NN while inserting data is worse than explicitly locating the point after the tree was built. Common trees like R-trees or kd-trees possess a structure for exploiting NN search as well as dividing the space into coverings. A further requirement is that this covering has to be disjoint. Thus we focused on the kd-tree family and found the local splitting technique of the LSD-tree to be the most suitable. The tree contains bucket nodes as leaves and directory nodes as internal nodes. The LSD-tree is a binary tree as the 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. As all buckets of the LSD-tree have a fixed maximum size of N, the data can be saved in a simple array with an allocated size of N. 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 N 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. Therefore, buckets after a split contain a list sorted according to the specific 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. One can think of the splitting as a hyperplane which separates the elements of the former bucket into two areas. 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. More precisely, we use circular incrementation with the dimensionality as upper bound of a counting variable from the root to the split bucket node to get the split-dimension, i.e. split-dimension = counted depth mod n. In another perspective, the LSD-tree partitions data of $\mathbb{R}^n_{0,+}$ 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. The difference between the LSD-tree and our `sLSD` is the lack of external directory nodes as we are constraining the problem for in-memory use cases. Additionally, the split decision of the classic LSD-tree is not rigidly predefined, so we have taken the liberty to do so. Lastly, our `sLSD` saves data of type Ω, but uses the function f to determine the position of the data.

Let us denote with $\text{depth}(e)$ the depth of any node e of an LSD-tree.

LEMMA 1. *Let b be an arbitrary leaf node. To determine the bounding cuboid of b we need to examine n ancestors of b.*

PROOF. *As each ancestor of b with depth* $\text{depth}(b) - j$

for $j \in \{1, \ldots, \text{depth}(b)\}$ *splits its descendants in dimension* $\text{depth}(b) - j \mod n$*, we have to backtrack exactly* n *ancestors of* b *to obtain split information about all dimensions. With this split information we can construct a minimal bounding cuboid containing all elements saved in the bucket node* b*.* □

Let $a \in \mathbb{R}^n_{0,+}$ be the NN of zero and B_a be the bucket node that includes a.

LEMMA 2. *The minimal bounding cuboid* $R := [0, a_1) \times \ldots \times [0, a_n)$ *of* B_a *divides the space into the regions* $D := [a_1, \infty) \times \ldots \times [a_n, \infty]$ *and* $G := \mathbb{R}^n_{0,+} \setminus (D \cup R)$*. Then* D *does not contain any skyline point.*

PROOF. *Because* a *is an element of* R *we follow that* $f(\Omega) \cap R \neq \emptyset$*. Thus for all vectors* $v \in D$ *we have* $a \prec v$*.* □

LEMMA 3. *The siblings of* B_a *as the descendants of their ancestors of depth at least* $\text{depth}(B_a) - n + 1$ *can contain skyline points.*

PROOF. *W.l.o.g. we assume* $\text{depth}(B_a) > n - 1$*. Let* p *be the ancestor of* B_a *with a depth of* $\text{depth}(B_a) - n$ *and let* l *be its left child. Then* l *is the ancestor of* B_a *with a depth of* $\text{depth}(B_a) - n + 1$*. The descendant directory nodes of* l *and* l *itself cannot have any split position greater than* p *if they have* p*'s split-dimension. Because of the definition of* B_a*, no directory node that is both ancestor of* B_a *and descendant of* p *has the split-dimension of* p*. Hence there is no leave node of* l *whose bounding cuboid is contained in* D *(of Lemma 2).* □

4. THE ALGORITHM

We study three different flavors of our algorithm resulting in (slightly) different running times:

- `SBQuick` with concurrent reverse quicksort-like merging of local skyline lists
- `SBFork` which excessively submits tasks to a fork/join pool [16] provided by Java 7's fork/join framework
- `SBSingle` without any concurrency and thus non-parallelized working on a single thread (We have implemented this flavor to show a fair comparison of already known non-concurrent algorithms)

For the underlying data structure we use two different flavors of LSD-trees:

- `sLSD-tree` - a simplified version of the LSD-tree without external nodes
- `spLSD-tree` - an enhanced version of the `sLSD`-tree with pruning methods described in Section 4.1

4.1 Pruning

Exactly as BNL++ [22], we can define a pruning condition for a bounded scoring function $f : \Omega \to \mathbb{N}^n_0$ by using the notation of $\text{height}_f := 1 + \sum_{i=1}^n \|\pi_i f\|_\infty$. Here $\|\cdot\|_\infty$ denotes the uniform norm, e.g. in our case $\|f\|_\infty = \max_{o \in \Omega} \|f(o)\|_{l_1}$ and $\|\pi_i f\|_\infty = \max_{o \in \Omega} \pi_i f(o)$. We further call the l_1 norm of the vector $f(x)$ the overall level of x. If x is a minimum or maximum of our relation $\prec$, we set

$$\text{pruningLevel}_f(x) := \begin{cases} 1 & \text{for } f(x) = 0 \\ \text{height}_f & \text{for } f(x) = \|f\|_\infty \end{cases}$$

Otherwise, we have

$$\text{pruningLevel}_f(x) := \sum_{i=1}^n \|\pi_i f\|_\infty - \min_{1 \leq i \leq n, \pi_i f(x) > 0} \left(\|\pi_i f\|_\infty - \pi_i f(x) \right)$$

For $x, y \in \Omega$ it can be shown that $x \prec y$ if

$$\text{pruningLevel}_f(x) \leq \|f(y)\|_{l_1}.$$

Therefore, while inserting data, we can retain the pruning level for each tuple, possibly lowering our previously noted minimum pruning level and thus giving us the possibility to drop data on insertion and/or drop already inserted data.

4.2 Populating the LSD-Tree

Given the assumption that our tree structure can keep the inserted data in the main memory, we just let the tree consume all of our data. The `spLSD-tree` only applies pruning to newly inserted data. That means, after lowering the pruning level, we do not scan through already inserted nodes, as our data structure is too slow for this purpose. The pruning level is saved globally, as multiple LSD-trees will be generated by both concurrent algorithms. Synchronizing this data sacrifices obviously parallel execution, as the data is accessed each time on data insertion. It could be beneficial to provide a more sophisticated read/write-lock than Java's `synchronized` keyword to gain slightly more performance.

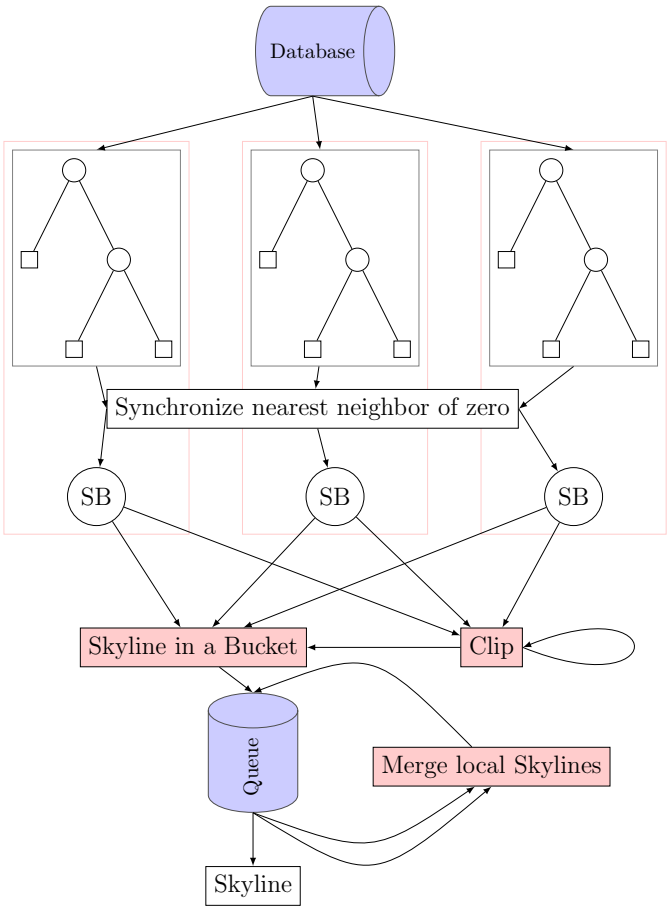
Algorithm 1 Skyline Breaker

```
function SB
    Q ← ∅
    N ← root
    while N is not a BucketNode do
        N ← N.left
    end while
    skyline(N) → Q
    a ← minarg_{a∈f(Ω)} ‖a‖_{l1}
    for i ← 0 to n do
        N ← N.parent
        skyline(N.right) → Q
    end for
    while N ≠ root do
        N ← N.parent
        if N.right is a BucketNode then
            skyline(N.right) → Q
        else
            v ← 0 ∈ ℝ^n_{0,+}
            d ← split-dimension of N
            v.(split-dimension of N) ← d
            clip(N.right, v, Q)
        end if
    end while
    return allocate(Q)
end function
```

4.3 The Skyline Breaker Algorithm

`SBQuick` and `SBFork` differ from the concept of how tasks are computed. Both algorithms generate an `sLSD-` or `spLSD-`tree for each thread. These trees are populated by items fetched from a sequential database input source. While

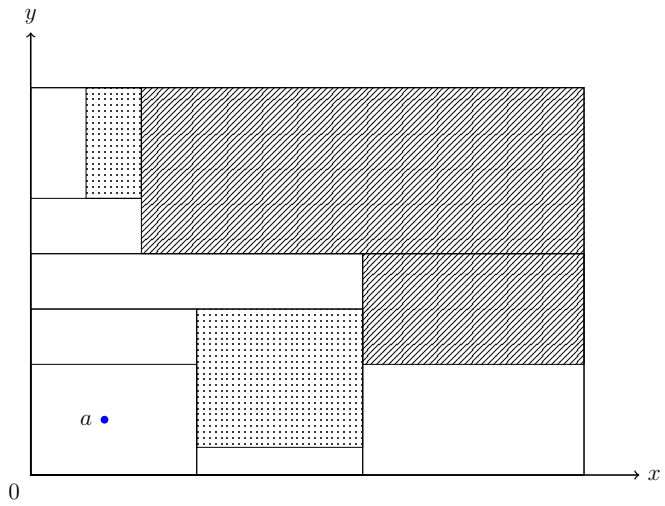
Figure 1: The schedule of the `SBFork`. Red colored boxes are tasks submitted to the `ForkJoinPool`



`SBQuick` lets each thread process its tree on its own, the `SBFork` gathers computation tasks and divides them with the help of the fork/join framework. The fork/join framework can be understood as a design pattern for concurrent DC programming. The main idea is that the whole problem can be split into tasks which are distributed under the worker threads, also simply called workers. Tasks of a fork/join framework are (recursive) functions that can generate new tasks (fork) and use the result of these tasks (join) to compose their solution. The workers keep their tasks in each one's scheduling queue, and poll this queue in a FIFO manner whenever they are not occupied. As tasks can recursively generate new tasks, queues will expand and shrink dynamically. Whenever a worker's queue is empty, it will try to take tasks from other workers by popping an element from the end of its queue. With this mechanism the fork/join framework tries to keep the workload balanced under all threads. For overview the schedule of `SBFork` is outlined in Figure 1.

A common task is to collect a bucket, i.e. to compute the local skyline of the data of a bucket (see Section 4.4) and add it for merging (see Section 4.6). A queue Q could be used to store local skyline sets. Tasks which contribute data with possible skyline points can then push their results into Q. Whenever Q is filled with at least two elements, a new task could join the local skylines, pushing the result back to Q.

Figure 2: We can ignore the dashed areas, but must not neglect the dotted ones!



Common denominator of all flavors is the start of the `SB` (Algorithm 1) immediately after the tree is filled with the input data. Firstly, we search the nearest neighbor of zero with respect to the l_1 norm in terms of bucket nodes. As our data is non-negative in all dimensions, locating this node is done by traversing to the outmost left leaf of the tree. We collect this bucket node B_a for local skyline computation and scan it for the nearest neighbor of zero. Let us denote this particular point by $a \in f(\Omega) \subset \mathbb{R}^n_{0,+}$. Now, having Lemma 1 in mind, we start to traverse the tree reversely, counting our steps upwards. Let us consider the model of n-dimensional cuboids for the nodes, where the directory nodes contain their children as cuboids. Each cuboid is parallel to all coordinate axes thanks to the split strategy of the LSD-tree. Thus each cuboid r has a corner point e_r which is nearest to zero with respect to the l_1 norm. As we will always come from the left child node, we just have to deal with the right node r. Until we have not counted upwards to the dimension n, at least one of e_r's coordinates is zero. This can be seen in Lemma 3. So we cannot discard r. Hence we need to traverse r and collect all of its buckets. When we have reached the number n with our counting, one of the child buckets of the right node could be discarded. That is because one of these child buckets might be contained in the region without skyline points considered in Lemma 2. Therefore, we start a new task that traverses the right node, done by the tree clipping, described in Section 4.5 and Algorithm 2. We have traversed the tree when we have reached the root node while climbing up the tree from the outmost left node. After that, `SBFork` has to wait for all tasks to get their jobs done, involving the procedures of Section 4.4 and Section 4.5. The last step is the allocation of the global skyline, as it is shown in Section 4.7. This step could be done as a periodically interim task or at the end of the schedule.

4.4 Skyline in a Bucket

Let us recall that in the step of insertion, buckets are sorted with respect to one direction when a split occurs. By using this split method, we get buckets which are mostly presorted with respect to the skyline order. The sorting

Algorithm 2 Clipping the Tree

```
function CLIP(N, v, Q)
    if N.left is a BucketNode then
        skyline(N.left) → Q
    else
        clip(N.left, v, Q)
    end if
    v' ← v
    d ← split-dimension of N
    v'.(split-dimension of N) ← d
    if a ⊀ v' then
        if N.right is a BucketNode then
            skyline(N.right) → Q
        else
            clip(N.right, v', Q)
        end if
    end if
end function
```

is advantageous for applying BNL or any other list-based skyline solver. Similar considerations with a full presort had led to the SFS [7] algorithm. In our case, we have chosen a simple inline BNL-based algorithm working directly on the array of the bucket. But we could have also used any other algorithm for computing the local skyline of a bucket.

4.5 The Tree Clip Algorithm

Algorithm 2 works with a DC-strategy which can be either implemented as a recursive function or as a `RecursiveTask` of the fork/join framework. We start with a directory node and the split information of its parent node. It is advisable to hold this data in a variable $p \in \mathbb{R}^n_{0,+}$ by setting the coordinate entry of p at the split-dimension to the split-position, while keeping all other coordinates at zero for a start. If we use the representation of cuboids again as in Section 4.3, the right child node r of our current directory node can be illustrated as a cuboid with the corner point $e_r = p$. So, while we traverse the children of our current directory node, we note down the split information in p until we have gained coordinates for all dimensions. With this information represented by the split point $p \in \mathbb{R}^n_{0,+}$, we can now check using the condition $a \prec p$ whether we can discard the right child node. If the condition holds, we can see that all elements contained in the cuboid r are dominated by a. Thus we can discard the node r, even if it is a directory node. Otherwise, we have to traverse the right node recursively, again denoting its split data. Figure 2 illustrates the nodes that can be discarded or must be looked up with regards to a given a and $n = 2$.

4.6 Local Skyline Combination

Combining two sets $V, W \subset f(\Omega)$ with local skyline points is done by iterating with a nested loop through both sets. The innermost loop compares $v \prec w$ and $w \prec v$ for all $v \in V, w \in W$ pairwise and removes dominated elements from their respective sets. At the end a new set with all remaining points is returned. Again, we could have used any existing skyline algorithm by using the union of both sets as input data.

4.7 Allocation of the Skyline

In `SBQuick`, each thread processes all tasks on its LSD-tree sequentially without concurrent interaction. Therefore, it has finally collected lists of local skyline points in a queue Q. In order to reduce Q to a list containing the global skyline, we apply a fold to Q with the local skyline list of B_a as the starting element and the algorithm of Section 4.6 as the function. On the other hand, `SBFork` instantaneously creates a new task for combining two local skylines whenever a task is done and Q is not empty.

5. THEORETICAL ASPECTS

5.1 Considerations about Pruning

Pruning can lead to an improvement of performance whenever correlative datasets are considered, as these datasets tend to have a high variance according to the overall level. Pruning of already inserted data is more expensive on the LSD-tree than on the list structure of BNL++. Hence a full pruning is not a good idea, as traversing and pruning the tree cannot be done simultaneously while inserting new tuples. Most effective pruning can be achieved by noting down the lowest pruning level of the already read data. This can be done by a full synchronization whenever a thread encounters a new tuple with a lower pruning level. Unfortunately, doing this while all threads are populating the LSD-tree might have a negative impact on parallelization.

5.2 Thread Frameworks

`SBQuick` refrains from depending on a complex thread-framework. The schedule of its threads is mostly independent resulting only in a sparse overhead due to thread usage. But with large data the possibility grows that threads get noticeably unbalanced workload. As a consequence, some of the threads might get unnecessarily unemployed while waiting for the other threads to get their jobs done. To provide a mechanism for balancing the workload among the threads, `SBFork` employs the fork/join framework. Java provides the `RecursiveAction` class for implementing the clipping of the tree (Algorithm 2). Smaller tasks can be submitted to the thread pool's task list as a `Runnable`. With a high number of threads, large data and a reasonably large maximum number of items per bucket, the `SBFork` should outperform the `SBQuick` by preventing idling. The price is a relatively large overhead with the creation of tasks and communication with the thread framework.

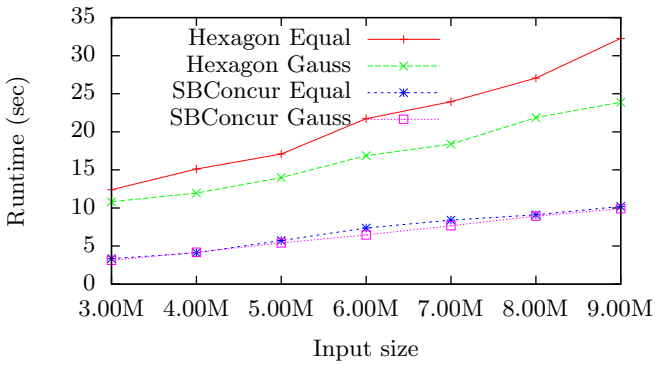
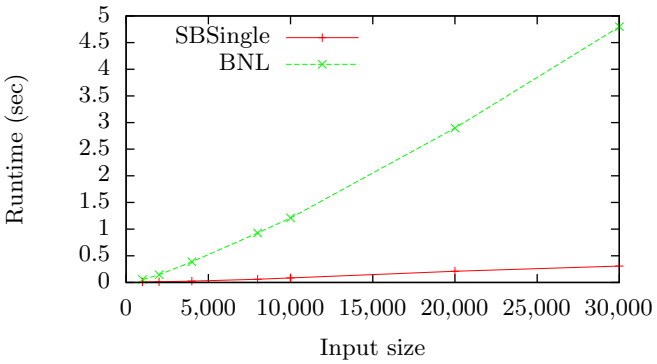


Figure 3: Gaussianly and uniformly distributed data, $f(\Omega) \subset [0, 100]^3 \times [0, 10]$

Figure 4: Single threaded evaluation

(a) Anti-correlated data, $f(\Omega) \subset [0, 10\,000]^4$



(b) Uniformly distributed, $f(\Omega) \subset [0, 100]^3 \times [0, 10]$

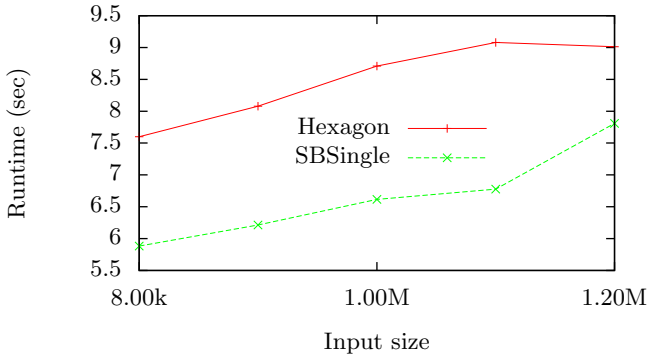
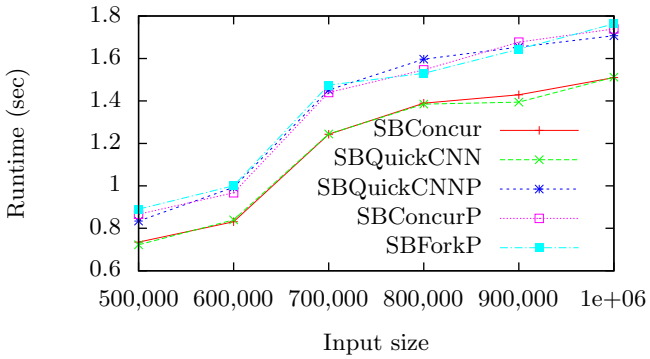
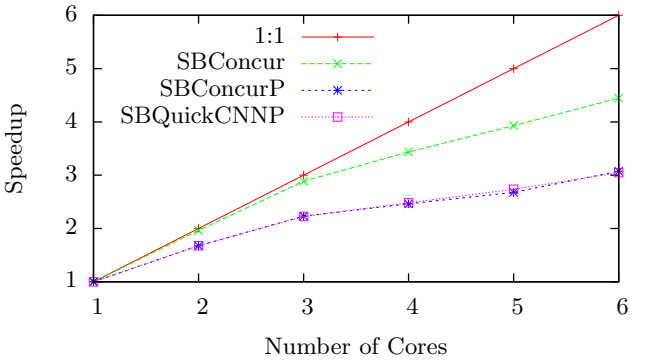


Figure 5: Gaussianly distributed data with $f(\Omega) \subset [0, 10\,000]^4$

(a) Variable Input Size



(b) Variable Number of Threads



5.3 Comparison with other algorithms

Now that the SB algorithms have been presented, we will consider a priori to the experiments in Section 6 with which kind of data sets the algorithms given in the experiments will well perform. Considering the simple data structure of BNL and its direct comparison of tuples, we can conclude that correlated data will not pose any challenge to BNL, as a huge quantity of tuples will already be discarded in its first loop. Moreover, high variances will improve the chance of finding a tuple with a relatively low pruning level compared to the average level of the data. Thus, given a distribution with large variance, BNL++ will have a good chance to prune the majority of already read data before attribute-wise comparison. Lastly, for Hexagon, we have to look at its special data structure, the BTG, that has to be built completely before the actual computation starts. A BTG is a Hasse diagram of the lattice (induced by $\prec$ and $=$) on $X := \{0, \dots, \pi_1 f(\Omega)\} \times \dots \times \{0, \dots, \pi_n f(\Omega)\}$ with $f(\Omega) \subseteq X \subset \mathbb{N}_0^n$. Thus, this graph represents the relation $\prec$ on a superset of all possible values. The creation of this structure needs at least $\mathcal{O}(|X|)$ time, so small values of $|X|$ are preferable. Contrary, $\mathcal{O}(|\Omega|)$ poses only a linear factor on computation time because tuples are saved in equivalence classes of the BTG, similar to our considerations in Subsection 5.4. When comparing these three algorithms with the ones presented above, we can see that a sequential SB will outperform neither BNL on correlated data nor Hexagon on small codomains of f around zero. Actually, we are interested in an algorithm whose running time is largely independent of distribution of data and robust against a large variance of dimensionality. Thus we like to outperform these algorithms, which have been studied ad nauseam, in the domains of their weaknesses.

5.4 Further Optimizations

For sake of simplicity, we do not cache the results of calls of f. Furthermore, if we neglect pruning, all tuples are inserted into the tree. So is it useful to keep evaluations of $f(x)$ for each tuple x in memory? With optimal conditions we would assume that f is injective, e.g.

$$I_f := \{x \in \Omega : \ \exists y \in \Omega \setminus \{x\} \ \text{ with } f(x) = f(y)\} = \emptyset.$$

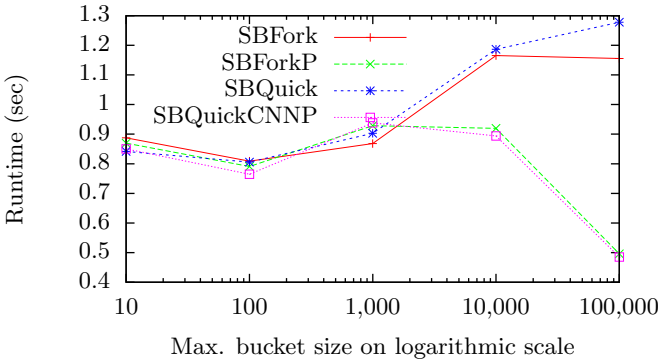
Then we have an identification $\Omega \simeq f(\Omega)$. Unfortunately, this is generally not the case (birthday paradox). Let us consider a bounded scoring function f with an integer-valued codomain. As the function is bounded, we have $|f(\Omega)| < \infty$. So there is a good chance that f is non-injective and the ratio $\frac{|I_f|}{|\Omega|}$ is above a certain limit. That is because one can expect that

$$\lim_{|\Omega| \to \infty} \frac{|I_f|}{|\Omega|} = 1 \text{ with fixed } |f(\Omega)| < \infty.$$

For this case we could work with the quotient space Ω / I_f, i.e. generate a mapping $\phi : f(\Omega) \to \mathcal{P}(\Omega)$ with $\phi(v) := \{x \in \Omega : f(x) = v\} \subset \Omega$. Thus we can apply our algorithm to the set $f(\Omega)$ and use ϕ to get the actual result. By the way, saving a lookup table of the form $\{(x, f(x))\}_{x \in \Omega}$ would

Figure 7: Measuring time for $f(\Omega) \subset [0, 10\,000]^4$ while varying N and keeping the input size $|\Omega| = 800\,000$ fixed.

(a) Correlated data



(b) Gaussianly distributed data

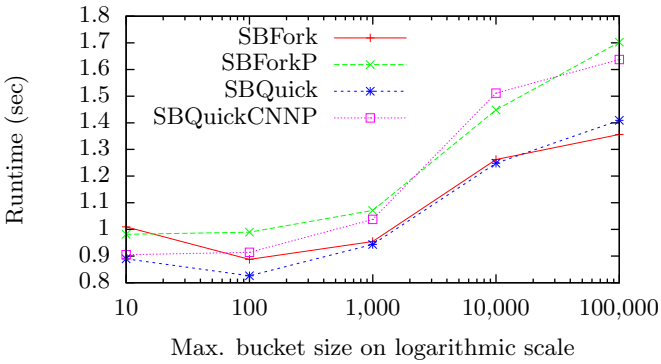
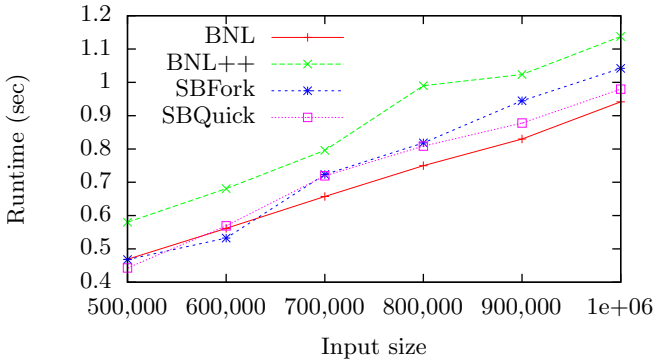
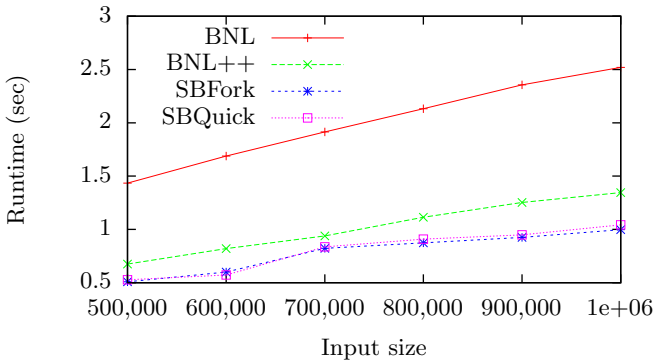


Figure 8: Benchmarking with $f(\Omega) \subset [0, 100]^3 \times [0, 10]$ againts BNL

(a) Correlated data



(b) Gaussianly distributed data



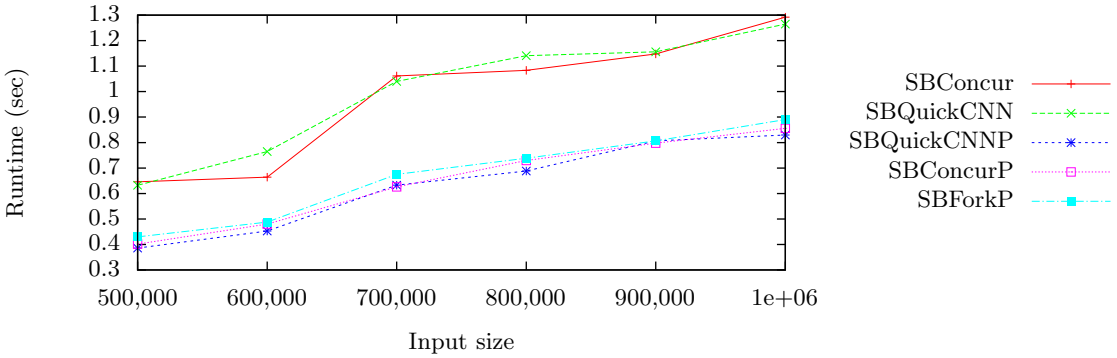
generally speed up our computation, especially when using pruning methods. Clearly, neglecting the pruning method, our algorithm could easily cope with more generally defined scorings of type $\Omega \to \mathbb{R}^n_{0,+}$, unlike Hexagon, which needs integer levels for building up the better-than-graph (BTG).

Lastly, the tree clipping (Algorithm 2) could be extended to use a list of known skyline points instead of a alone (a as defined in Section 4.3). For example, any point $b \in f(\Omega)$ with $\|a\|_{l_1} = \|b\|_{l_1}$ belongs to a skyline tuple. Thus we could build up a partial skyline set to provide a higher chance of clipping.

6. EXPERIMENTS

In this section we evaluate our algorithm against known frameworks to show its efficiency. Our algorithm is tested on synthetically generated data sets as suggested by Börzsönyi et al. [5]. The test data varies in the dimensionality n, the cardinality of the input size $|\Omega|$, expected value and variance. We used anti-correlated and correlated data as well as uniformly and Gaussianly distributed data with regards to f. All shown algorithms are implemented in Java. They are agnostic towards the type of tuples they fetch from our emulated database connection. That means each tuple has to be evaluated by a bounded scoring function $f : \Omega \to \mathbb{N}^n_0$. For simplicity, the identity $f = id : \mathbb{N}^n_0 \to \mathbb{N}^n_0$ was used as scoring function with a small computation drawback. The implementation of BNL shown here is based on [5], but with a suitably large window containing at most 100 000 tuples. So it should be guaranteed that the algorithm does not suffer from spare memory usage, i.e. only few passes are necessary to complete the full skyline. To simplify reading we combined `SBFork` and `SBQuick` to `SBConcur` whenever their graphs have only slight aberrations. Moreover, we have suffixes to the captions whenever a particular flavor is used. The suffix `P` implies that pruning methods are used on the tree generation. The suffix `CNN` applied to `SBQuick` means that the nearest neighbor of zero is synchronized between all threads before the `SB`-algorithm starts. Lastly, `P` and `CNN` can be combined to `CNNP`. Let us start the evaluation without concurrency, showing how `SBSingle` performs. We start off with Hexagon. When competing with Hexagon, we have to choose a small cardinality of the codomain as it generates the full BTG. We can see in Figure 4b that our algorithm will be easily surpassed by Hexagon with a small codomain when augmenting input size. Following our considerations in Section 5.4, enabling pruning and working with equivalence classes could provide a benefit in this situation. For Figure 4a it is interesting to see how badly BNL performs on anti-correlated data. We see that the `SBSingle` is very robust against different distributions. It is easy to conclude that this property is carried over to its concurrent siblings. Now let us see how the `SB` algorithm competes when deploying concurrency. We first show a comparison between the

Figure 6: Correlated data, $f(\Omega) \subset [0, 10\,000]^2$



different flavors. When working with non-correlated data and a relatively low cardinality of I_f, the cost of pruning is higher than its benefits, as shown in Figure 5a. A classic regression curve reflecting Amdahl's law can be seen in Figure 5b. The figure shows the evaluation of different `SB`-flavors with a fixed input size but different numbers of threads (represented by the x-axis). For each algorithm the running time is subsequently divided by the running time the algorithm took for running with only one thread, i.e. without parallelization. An optimal algorithm would take half of the time if running with two threads, one third of the time if running with three threads, and so forth. Thus an optimal result would be achieved if the curve of an algorithm followed the line $x = y$. We can also see that the parallelization of `P`-flavors suffers especially from the synchronized lookup of the pruning variables. On the other hand, pruning can be beneficial when working with correlated data on small codomains, as shown in Figure 6. Until now, we have fixed the maximum number N of elements a bucket can contain at 100. Now we want to see how this number can affect skyline computation of the concurrent `SB` algorithm flavors. At first glance it is interesting to see how pruning techniques benefit from an increasing N in Figure 7a. We have to keep in mind that an increase of N decreases the depth of the LSD-tree towards the limit where the LSD-tree contains a single bucket. Whenever we have this case, our approach resembles the characteristics of BNL. We can see by Figure 7 that $N = 100$ is an optimal value for an input size of $|\Omega| = 800\,000$. Lastly, we like to catch a glimpse on how performant the `SBConcur`-family is when competing with BNL, BNL++ and Hexagon. Even on correlated data as in Figure 8a our strategy can keep up with BNL. With Gaussianly distributed data (Figure 8b) our algorithm has a clear advantage over both BNL and BNL++. Figure 3 clearly shows how Hexagon is loosing speed with higher input values. This benchmark was evaluated with both equally and Gaussianly distributed data while keeping the codomain in the same bounds. For computation we used a machine running linux (Debian Wheezy) with 24 GB RAM and a processor of the type "Intel(R) Xeon(R) CPU E5540" with four cores. With Intel's Hyper-Threading Technology we could spawn up to eight threads working in parallel. Only for Figure 5b, we used a personal computer running Arch Linux 2013.4 with 8 GB RAM and an "AMD FX(tm)-6100" processor with six cores. The technical prototype of our algorithm is available for re-evaluation or implementation under a BSD-license at http://developer.berlios.de/projects/skylinebreaker.

7. CONCLUSION

We have studied a concurrent algorithm for skyline computation. The dimension-robustness of the LSD-tree granted us the ability to evaluate complex features (e.g. high dimensionality of the codomain of f). Additionally, the disjoint splitting lead to the SB algorithm providing the feature to clip entire directory nodes without further evaluation of the data in depth. As our proposed clipping method is a DC algorithm, we could spawn cuncurrent tasks on subsets of the data to calculate independently to each other a local skyline set. With this strategy we sought to apply parallel execution of the main algorithm exhaustively. Supplementary pruning can be beneficial when coping with correlated data and large variance of the codomain. By splitting up tree population and traversing the tree we got two different approaches for concurrent programming of the `SB`-algorithm, `SBFork` and `SBQuick`. The former evolved by interspersing concurrent structures powered by the fork/join framework, the latter was formed by applying a MapReduce-similar strategy on the sequential `SB` algorithm. In the evaluation we addressed the stabilitiy of `SB` with different distributions and highlighted concurrent efficiency. Our algorithm showed its strengths and weaknesses with regards to well-known skyline algorithms.

8. OUTLOOK

A benchmark of `SB` with parallel implementations of common skyline algorithms is yet to come. Performance gains could be achieved by proposed optimistic parallelism methods [23] and more fine-grained synchronization. With the advent of parallel databases, next-generation databases possess the ability to deliver results in a parallel manner so that we can avoid the sequential step of fetching the data. It could be interesting to see how performance can be gained by parallel data input. Furthermore, aspects like top-k evaluation, a variation of `SB` as an online algorithm and externalization of memory, have not been discussed in this paper.

Acknowledgement

We show our gratitude to Markus Endres for providing us his data generator library and the implementations of BNL, BNL++ and Hexagon. Moreover, we are grateful to Roland Glück, Bastian Kuhn and Manuel Mulzer for proofreading this paper.

References

[1] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman. Parallel skyline queries. In A. Deutsch, editor, *ICDT*, page 274–284. ACM, 2012. ISBN 978-1-4503-0791-8.

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

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9): 509–517, Sept. 1975. ISSN 0001-0782. doi: 10.1145/361002.361007.

[4] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[5] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proceedings of the 17th International Conference on Data Engineering*, page 421–430, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1001-9.

[6] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. On High Dimensional Skylines. In Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, and C. Böhm, editors, *EDBT*, volume 3896 of *Lecture Notes in Computer Science*, page 478–495. Springer, 2006. ISBN 3-540-32960-9.

[7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting: Theory and Optimizations. In M. A. Klopotek, S. T. Wierzchon, and K. Trojanowski, editors, *Intelligent Information Systems*, Advances in Soft Computing, page 595–604. Springer, 2005. ISBN 3-540-25056-5.

[8] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[9] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB J.*, 16 (1):5–28, 2007.

[10] J. L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, 1988.

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

[12] A. Henrich, H.-W. Six, and P. Widmayer. The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects. In P. M. G. Apers and G. Wiederhold, editors, *VLDB*, page 45–53. Morgan Kaufmann, 1989. ISBN 1-55860-101-5.

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

[14] 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.

[15] H. T. Kung, F. Luccio, and F. P. Preparata. On Finding the Maxima of a Set of Vectors. *J. ACM*, 22(4):469–476, 1975.

[16] D. Lea. A Java fork/join framework. In *Proceedings of the Java Grande 2000 Conference*, page 36–43, 2000.

[17] H.-X. Lu, Y. Luo, and X. Lin. An Optimal Divide-Conquer Algorithm for 2D Skyline Queries. In L. A. Kalinichenko, R. Manthey, B. Thalheim, and U. Wloka, editors, *ADBIS*, volume 2798 of *Lecture Notes in Computer Science*, page 46–60. Springer, 2003. ISBN 3-540-20047-9.

[18] T. Mattson and M. Wrinn. Parallel programming: can we PLEASE get it right this time? In L. Fix, editor, *DAC*, page 7–11. ACM, 2008. ISBN 978-1-60558-115-6.

[19] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *SIGMOD Conference*, page 467–478. ACM, 2003. ISBN 1-58113-634-X.

[20] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-k. Larson, and B. C. Ooi, editors, *VLDB*, page 253–264. ACM, 2005. ISBN 1-59593-177-5.

[21] T. Preisinger and W. Kießling. The Hexagon Algorithm for Pareto Preference Queries. 2007.

[22] T. Preisinger, W. Kießling, and M. Endres. The BNL++ Algorithm for Evaluating Pareto Preference Queries. In *In Proceedings of the Multidisciplinary Workshop on Advances in Preference Handling (ECAI)*, 2006.

[23] J. Selke, C. Lofi, and W.-T. Balke. *Highly Scalable Multiprocessing Algorithms for Preference-Based Database Retrieval*. Springer, 2010. ISBN 978-3-642-12098-5. doi: 10.1007/978-3-642-12098-5_19.

[24] B. Stefan, K. D. A., and K. Hans-Peter. The X-tree: An Index Structure for High-Dimensional Data, 1996.

[25] R. Torlone and P. Ciaccia. Finding the Best when it's a Matter of Preference. In *SEBD*, page 347–360, 2002.

[26] L. Woods, G. Alonso, and J. Teubner. Parallel Computation of Skyline Queries. In *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '13, page 1–8, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4969-9. doi: 10.1109/FCCM.2013.18.

[27] Y. Yuan, X. Lin, Q. Liu, W. W. 0011, J. X. Yu, and Q. Zhang. Efficient Computation of the Skyline Cube. In *VLDB*, page 241–252. ACM, 2005. ISBN 1-59593-177-5.