SkyMap: A Trie-Based Index Structure for High-Performance Skyline Query Processing

Joachim Selke and Wolf-Tilo Balke

Institut für Informationssysteme
Technische Universität Braunschweig
Braunschweig, Germany
{selke,balke}@ifis.cs.tu-bs.de

Abstract. Skyline queries have become commonplace in many applications. The main problem is to efficiently find the set of Pareto-optimal choices from a large amount of database items. Several algorithms and indexing techniques have been proposed recently, but until now no indexing technique was able to address all problems for skyline queries in realistic applications: fast access, superior scalability even for higher dimensions, and low costs for maintenance in face of data updates. In this paper we design and evaluate a trie-based indexing technique that solves the major efficiency bottlenecks of skyline queries. It scales gracefully even for high dimensional queries, is largely independent of the underlying data distributions, and allows for efficient updates. Our experiments on real and synthetic datasets show a performance increase of up to two orders of magnitude compared to previous indexing techniques.

1 Introduction

Skyline queries, introduced in [2], quickly found its way into a widespread range of data management applications. Soon after the first skyline algorithms have been presented, emerging fields like e-shopping or location-based services used the economic intuitiveness of Pareto-optimal result sets, e.g., for deriving all reasonable product alternatives [15,8]. The basic idea is to filter out all those items (database tuples) dominated in terms of usefulness by other items, i.e., there is no utility function declaring a dominated item as best choice. For example, when trying to find an inexpensive hotel close to the beach, overpriced inland hotels can safely be excluded.

In a brief period of time, several important problem settings building on the original skyline paradigm have been identified and individual solutions have already been proposed, e.g., [3], [14], or [11]. However, all these algorithms rely on specialized techniques, whereas the widespread applicability of skyline queries also raises the question of how these queries can be answered efficiently in general-purpose database systems. This especially sparked interest in the use of indexing techniques to boost skyline query performance (e.g., [11], [6], [16], or [9]). Indexes are indeed the key for broad database support for efficient skyline compution. In fact, [9] shows that a wide variety of special skyline queries (k-dominant skylines, skybands, subspace skylines, etc.) can be supported using a single index structure. But although indexes can dramatically speed up

A. Hameurlain et al. (Eds.): DEXA 2011, Part II, LNCS 6861, pp. 350–365, 2011.
© Springer-Verlag Berlin Heidelberg 2011

retrieval, they of course also introduce maintenance costs and tend to quickly degenerate on higher dimensional data. *The task thus is to design a robust index structure that (1) enables high-performance skyline processing, (2) is easy to maintain, and (3) gracefully scales with data dimensionality.*

In this paper we propose SkyMap, an innovative trie-based index structure for skyline query support. In a nutshell, SkyMap adopts a recursive grid-like space partitioning approach, which facilitates efficient navigation. In contrast to traditional data-driven space partioning with the inherent danger of degeneration over time, SkyMap takes the problem domain into account. Since skyline computation builds on ranks rather than absolute scores, any data set can be transformed spreading data evenly over each dimension. Thus, an efficient data independent space partitioning scheme without degeneration problems can be used, and the trie generally stays balanced. In particular, we will show that for each data set there exists only a single unique SkyMap making the resulting trie structure independent of the order in which data is inserted or deleted.

Moreover, data is only contained in the trie's leaf nodes, which fosters efficient maintenance operations. Especially, expensive rebalancing operations are avoided. Navigation within the SkyMap index is particularly efficient by relying on inexpensive bitwise operations only. The index also closely controls each node's fan-out by limiting the split factors, to optimize the trie's depth-to-width ratio. Thus, the index's degradation into a linear list is effectively prevented.

We extensively evaluated SkyMap on different data sets and compared its performace to the current state-of-the-art indexing schemes: approaches based on quadtrees (OSP-SPF [16] and BSkyTree [7]) and on UB-trees (ZB-tree [9]). Although these techniques show their strength in different aspects (higher pruning power vs. improved maintenance), SkyMap outperforms them on all counts. We show that our trie-based indexing scheme gains an order of magnitude in query performance across the board, for skyline maintenance even up to two orders of magnitude.

2 Related Work

The investigation of skyline processing started out with list-based approaches like the well-known BNL algorithm [2] and at first focused on algorithmic aspects and sophisticated heuristics like presorting [4] or limiting [1]. However, the necessity for efficient indexing schemes was soon recognized. Indexes allow for efficient pruning, hence large parts of the database can immediately be excluded from costly dominance tests. The first approaches featured efficient bitmap indexes [5] and R-trees with nearest neighbor search [6] or branch-and-bound pruning [11]. The major problem with these approaches was that the underlying data structures did not lend themselves readily for adoption to skylining problems. Since overlapping regions in multi-dimensional index structures proved to pose severe performance penalties, recent approaches focus on strictly disjoint space partionings or efficient one-dimensional indexing using space-filling curves.

The ZB-tree [9] is based on the UB-tree, where each data point is mapped to its Z-address, which in turn provides a one-dimensional key for B-tree indexing. The benefits of this approach for skyline processing are twofold. On one hand, through the Z-addresses the B-tree imposes a presorting on the data, which can be exploited for dominance tests: No database item can dominate any item having a lower Z-address.

On the other hand, the regions covered by each tree node can easily be estimated (upper bound) without keeping track of minimum bounding rectangles. A major drawback of the ZB-tree approach is that regions may overlap, which hampers effective pruning. Moreover, the maintenance of B-trees is rather expensive in case of frequent updates, in particular due to rebalancing operations caused by node underflows. Still, the actual degeneration of the index structure over time is not a problem.

OSPSPF [16] and the BSkyTree [7] focus on the problem of overlapping regions and improve performance by disjoint space partitioning. Both approaches rely on quadtrees and basically are alternative implementations of the same idea. Each tree node stores a single data item, which is used to split the underyling space with respect to each dimension. The actual performance of this data-driven partitioning scheme is strongly dependent on finding optimal splitting points. While a careful initial bulkloading will lead to a perfectly balanced tree structure, frequent updates cause the index' performance to deteriorate quickly. Moreover, since all nodes store actual data, deletions either ruin cache efficiency when performing lazy deletions or force expensive reorganization operations.

In summary, we are forced to conclude that there is no single index structure offering superior pruning power, easy maintenance, and graceful scaling.

3 Preliminaries

Henceforth, we consider a set of n database items, where each item's utility can be evaluated with respect to scorings over d criteria. Without loss of generality we assume that each score lies within the interval $[0,1)$ and that higher values are better. Then, the database can be represented as a set $A \subset [0,1)^d$ of size n.

Skylines captures the intuitive idea of Pareto optimality. Formally, given two points $x = (x_1, \dots, x_d)$ and $y = (y_1, \dots, y_d)$, the point x is said to *dominate* y (denoted by $x \succ y$) if and only if $x \neq y$ and $x_i \geq y_i$, for each dimension i. Furthermore, x and y are said to be *incomparable* (denoted by $x \parallel y$) if and only if neither $x = y$, nor $x \succ y$, nor $x \prec y$. We write $x \succ_i y$ if $x_i > y_i$. The skyline of a data set consists of exactly those items that cannot be ruled out by means of Pareto dominance. Formally, the *skyline* of a point set A is the set $\mathscr{S}(A) = \{x \in A \,|\, \text{there is no } y \in A \text{ such that } y \succ x\}$. Characteristic properties of a data set A are its dimensionality d, its cardinality n, and its skyline size $s := |\mathscr{S}(A)|$.

As a running example, consider the data set depicted in Fig. 1. Each point corresponds to a database item, scored with respect to two different criteria. We see that items A, B and C can safely be removed from further consideration since each is dominated by some other item (A is dominated by F, E, and D; B is dominated by F, E, and D; and C is dominated by D). The remaining items D, E, and F form the skyline.

4 The SkyMap Approach

The design goals for a skyline-centered indexing technique can be divided into two categories: boosting performance and minimizing maintenance overhead. For *boosting performance*, the index structure should be well-balanced, avoid overlaps between data regions, and control the nodes' fan-out degree to prevent degradation into a linear list. For *minimizing the maintenance overhead*, all data should be collected in leaf nodes,

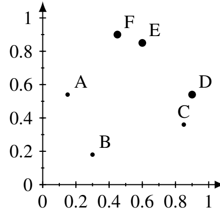


Fig. 1. Our running example

whereas all internal nodes should be purely navigational. Expensive reorganizations of the index structure must be avoided.

Although these goals sound contradictory, we show that it is indeed possible to implement them within a single data structure. Looking at the overlap problem, it quickly becomes clear that disjoint space-partitioning is needed. Combining this partitioning with our requirement of leaf-only data storage while minimizing reorganizations definitely calls for a data-independent partitioning scheme. Of course, such a data-independent partitioning would hurt the index structure's balancedness whenever the data distribution is skewed in any dimension. But can we guarantee evenly spread data for skyline computations? Interestingly, we can.

This is because the notion of Pareto dominance, and thus the base of skyline computation, does only rely on the actual score values in so far as they introduce a dominance ranking between database items. That means: As long as the relative ranking with respect to each dimension stays intact, we may arbitrarily modify all absolute score values. In particular, we may apply a transformation to each individual dimension that spreads database items evenly. This class of mathematical transformations is known as *copulas* [10]. Copulas are computationally inexpensive and are typically applied for correlation analysis. The following (intuitive and easy-to-prove) lemma shows that computing the skyline of a copula-transformed data set will indeed yield the data set's original skyline.

Lemma 1. *Let A be a d-dimensional data set, $f_1,\ldots,f_d : [0,1) \to [0,1)$ strictly monotonic increasing functions, $f(x) := (f_1(x_1),\ldots,f_d(x_d))$, and $f(A) := \{f(x) \,|\, x \in A\}$. Then, $x \in \mathscr{S}(A)$ if and only if $f(x) \in \mathscr{S}\big(f(A)\big)$, for any x.*

4.1 Tries

To satisfy our design goals, we chose to base the SkyMap index on tries. The term *trie* refers to all tree data structures that perform a data-independent disjoint space-partitioning and use (typically binary) strings for navigation [13].

To give an example, Fig. 2 illustrates the difference between a binary search tree (BST) and a 3-digit binary trie (3BT), when it comes to storing a given set of numbers. While there is a one-to-one mapping between nodes and data items in the BST, the 3BT differentiates between internal nodes, which are solely used for navigational purposes, and leaf nodes, which store the actual data. Moreover, the BST is constructed only by performing ordinal comparisons between numbers, whereas the 3BT exploits the

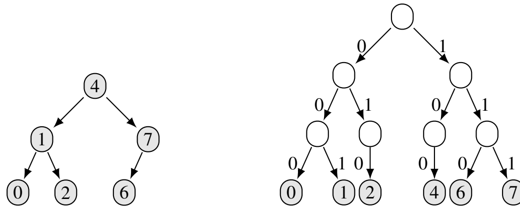


Fig. 2. A binary search tree and a 3-digit binary trie storing the set $\{0,1,2,4,6,7\}$

numbers' binary representation. Finally, there are many different BSTs storing the data set shown in Fig. 2, whereas the 3BT is unique.

Quadtrees are the multidimensional extension of BSTs, while our SkyMap is a multidimensional extension of binary tries, which additionally provides efficient algorithms for dominance checks. The regular and predictable structure of tries provides high memory locality and enables efficient updates of the stored data set—two key ingredients of SkyMap. It also heavily exploits the binary representation of data points. Henceforth, we indicate score values written in binary by the prefix $\mathscr{B}$, e.g., $0.75 = \frac{1}{2} + \frac{1}{4} = \mathscr{B}0.11$.

For easy presentation and without loss of generality, we assume in the following that all of A's points are distinct.

4.2 Z-Addresses, Z-Regions, and Z-Subregions

In Section 3 we assumed without loss of generality that all data points are located in $[0,1)^d$. Therefore, the i-th coordinate value of the point $x = (x_1, \dots, x_d)$ can be represented as $x_i = \mathscr{B}0.\alpha_{i,1}\alpha_{i,2}\cdots$, where $\alpha_{i,1}, \alpha_{i,2}, \ldots \in \{0,1\}$ are x_i's binary digits. By interleaving the bits of all of x's coordinate values, the *Z-address* of x, denoted by $\mathrm{Z}(x)$, can be derived [12]:

$$\mathrm{Z}(x) = \mathscr{B}0.\alpha_{1,1}\cdots\alpha_{d,1}\alpha_{1,2}\cdots\alpha_{d,2}\cdots.$$

In general, the j-th bit of a Z-address is determined by the $\big(((j-1)\operatorname{div} d)+1\big)$-th bit of the $\big(((j-1) \bmod d)+1\big)$-th coordinate value. For example, the Z-address of the three-dimensional point $(0.125, 0.75, 0.5) = \mathscr{B}(0.001, 0.11, 0.1)$ is $\mathscr{B}0.0110101$.

When reversing the above situation, also each Z-address uniquely defines a d-dimensional point in the (upper open) unit hypercube. We will use this property to assign a rectangular region in space to each finite binary sequence as follows: Given a number $y = \mathscr{B}0.\beta_1\beta_2\cdots \in [0,1)$ and a non-negative integer r, we define the r-th *Z-region* of y, denoted by $\mathrm{ZR}_r(y)$, to be the set of all points whose Z-address (in binary) begins with the sequence $\mathscr{B}0.\beta_1\beta_2\cdots\beta_r$. That is,

$$\mathrm{ZR}_r(y) = \Big\{x \in [0,1)^d \;\Big|\; \mathrm{Z}(x) = \mathscr{B}0.\beta_1\cdots\beta_r\gamma_{r+1}\gamma_{r+2}\cdots\Big\}.$$

In general, any Z-region is a lower closed and upper open hyperrectangle.

Every Z-region can naturally be partitioned into smaller Z-regions as follows: Given a number $y = \mathscr{B}0.\beta_1\beta_2\cdots \in [0,1)$ as well as two non-negative integers r and b, then for any $\gamma_1, \ldots, \gamma_b \in \{0,1\}$, the $(r+b)$-th Z-region of the number $\mathscr{B}0.\beta_1\cdots\beta_r\gamma_1\cdots\gamma_b$ is

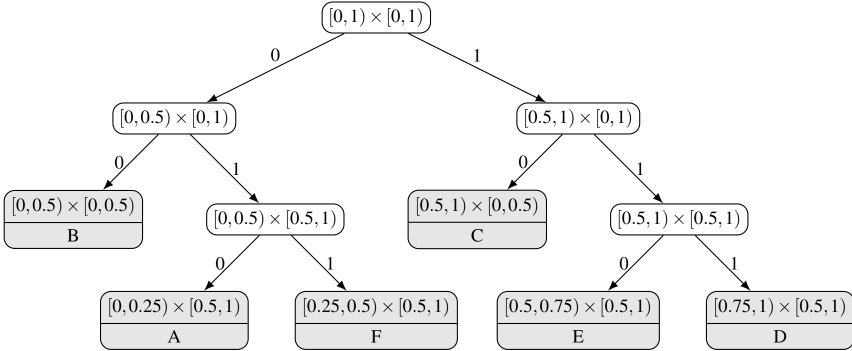


Fig. 3. A SkyMap ($b = 1$, $c = 1$) indexing our example data set

a subset of $\mathrm{ZR}_r(y)$. In total, there are 2^b such subsets, which are mutually disjoint. This way, for any b, the Z-region $\mathrm{ZR}_r(y)$ can be divided into 2^b partitions. In the following, we will refer to these partitions as $\mathrm{ZR}_r(y)$'s *Z-subregions* of degree b. For example, in two-dimensional space, the degree-2 Z-subregions of the Z-region $\mathrm{ZR}_0(0) = [0,1)^2$ are $\mathrm{ZR}_2(0) = [0,0.5) \times [0,0.5)$, $\mathrm{ZR}_2(0.25) = [0,0.5) \times [0.5,1)$, $\mathrm{ZR}_2(0.5) = [0.5,1) \times [0,0.5)$, and $\mathrm{ZR}_2(0.75) = [0.5,1) \times [0.5,1)$.

4.3 SkyMap and Its Basic Operations

In this section, we build the skeleton of SkyMap. For this task, we turn the concepts just presented into a recursive scheme for space decomposition.

A SkyMap index is a trie that consists of internal nodes and leaf nodes: *Leaf nodes* store a list of at least one but at most c data points, where c is some integer we will refer to as the *capacity* of a leaf node. We assume that all leaf nodes have the same capacity. *Internal nodes* store an array of exactly 2^b pointers to its child nodes, which are indexed by the numbers 0 to $2^b - 1$, where b again is an integer parameter shared by all nodes. We will refer to b as the *split degree* of an internal node (measured in bits). Null pointers are generally allowed, but empty nodes are not permitted.

The SkyMap index has been designed to resemble the recursive splitting process of Z-regions into all its 2^b Z-subregions of degree b. In particular, each node represents some Z-region, where the root node corresponds to the Z-region $\mathrm{ZR}_0(0) = [0,1)^d$.

Fig. 3 shows a normalized version of our example database and a corresponding SkyMap indexing it. We set $b = 1$, which means that the data space is recursively split along a single dimension each. The respective dimension changes from level to level. The capacity c is set to 1, i.e., each leaf node stores a single point. To illustrate the effect of these parameters, Fig. 4 shows an alternate SkyMap with $b = 2$ and $c = 2$.

In contrast to quadtree-based approaches, where nodes may have up to 2^d children, in a SkyMap index the number of children of each node can easily be controlled by setting the parameter b accordingly. That way, degradation of the trie structure to a linear list can be avoided in higher dimensional spaces. On the other hand, b can be chosen large enough to enable a most effective space division and pruning. Moreover,

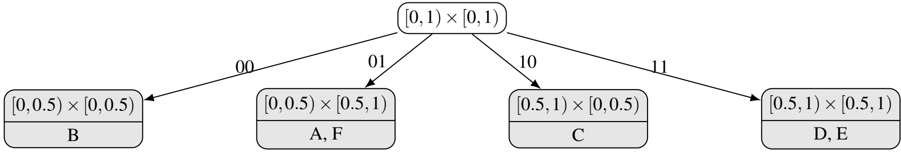


Fig. 4. A SkyMap ($b = 2$, $c = 2$) indexing our example data set

by tuning the leaf capacity c, one can avoid excessive ramification of the trie but still receive the benefits of indexing, thus exploiting the full potential of memory caches.

The most important operations required for building and maintaining an index are *bulkloading* of entire data sets as well as *inserting* and *deleting* individual items. As we will show next, these operations can be performed highly efficient in a SkyMap index.

In SkyMap, bulkloading of a given data set A is performed in top-down fashion. After presorting the data set by Z-addresses, we recursively split the data set with respect to Z-address prefixes of length b and create internal nodes (if the number of remaining points is larger than c) or leaf nodes (otherwise). During this process, we completely avoid expensive multi-dimensional point comparisons and only rely on cheap integer comparisons and bit-shifting operations. The complete algorithm is shown in Fig. 5.

To insert a new point p into an existing SkyMap (cf. Fig. 5), we traverse the trie structure according to p's Z-address and add p to a matching leaf node. If the leaf node already is filled to its maximum capacity c, we replace it by a new internal node. Then, the insertion process continues at the new node. Again, we do not perform explicit point comparisons but only need to perform integer comparisons and bit shifts. Moreover, our insertion procedure does not require expensive rebalancing operations.

In a similar fashion, points can be removed from an existing SkyMap index: First, find the corresponding leaf node *leaf* and delete the point. In case *leaf* gets empty by this operation, also remove *leaf* from the index. If necessary, recursively repeat this cleanup process with all parental internal nodes. Moreover, the deletion algorithm keeps track of how many points are located below each internal nodes it visits during the cleanup process. If an internal node is detected that covers only c points or less, then this internal node is replaced by a corresponding leaf node, thus reducing the SkyMap's depth to the minimum possible value. Due to space limitations, we omit the pseudocode.

4.4 Analysis

Having introduced the basic structure of the SkyMap index, we now demonstrate that it lives up to its promise. Recall the two main requirements: boosting performance and minimizing maintenance overhead. By designing simple insert and delete operations, which only affect a very small number of nodes and do not perform any expensive rebalancing or restructuring, we have already fulfilled our second requirement of easy maintenance. At the same time, due to the use of copulas for normalizing our data set (i.e., distributing it equally across each dimension; see Lemma 1), we are able to achieve a homogeneous distribution of data points across the whole SkyMap tree (given that data dimensions do not exhibit correlation or anti-correlation at an extreme degree).

```
function BULKLOAD(A)
    Presort A by Z-addresses; let P and Z be the resulting lists of points and Z-addresses
    root ← BULKLOAD(P, Z, 0, |A| − 1)

function BULKLOAD(P, Z, from, to)
    size ← to − from + 1
    if size ≤ c then return a new leaf node containing P[from, ..., to]
    else
        node ← a new empty internal node
        Z' ← a copy of Z, in which the first b bits have been removed from each entry
        pos ← from
        zbFrom ← the first b bits of Z[from]
        while pos ≤ to do
            zbPos ← the first b bits of Z[pos]
            if zbPos ≠ zbFrom then
                node.children[zbFrom] ← BULKLOAD(P, Z', from, pos − 1)
                from ← pos
                zbFrom ← zPos
        node.children[zbFrom] ← BULKLOAD(P, Z', from, to)
        return node

function INSERT(p)
    if root = null then root ← a new leaf node containing only p
    else
        z ← p's Z-address
        INSERT(p, z, root, 0)

function INSERT(p, z, node, depth)
    zb ← the first b bits of z
    if node is an internal node then
        if node.children[zb] = null then node.children[zb] ← a new leaf containing only p
        else
            z' ← z without its the first b bits
            INSERT(p, z', node.children[zb], depth + b)
    else if node is a leaf node containing less than c points then Add p to node
    else
        node' ← a new empty internal node
        for each point q contained in node do
            qz ← q's Z-address without its first depth bits
            INSERT(q, qz, node', depth)
        INSERT(p, z, node', depth)
        Replace node by node' in the SkyMap index
```

Fig. 5. Creating a SkyMap by bulkloading and inserting a new point into an existing SkyMap

We now show that SkyMap also possesses two key ingredients of high-performance index structures: immunity against degradation due to database updates and logarithmic depth on average.

Lemma 2. *Let the parameters b and c be fixed. Then, for any data set A, there is a unique SkyMap indexing A, modulo sort order of points within leaves.*

Proof. Assume that there are two different SkyMap indexes storing the data set A. Since by design each point is always stored in a leaf node along its Z-address path, to be different modulo sort order of points within leaf nodes, one of the SkyMap indexes must contain a leaf node that is not present in the other one. Let N be this node. We assume without loss of generality that N is located within the first index. Since any point contained in N must also be present in some leaf node of the second index, there must be a leaf node M in the second index at the location of one of N's parental internal nodes. As M can contain at most c points and points are placed in the index along their

Z-addresses, also the subtrie rooted at the internal node corresponding to M in the first trie can contain at most c points, a situation which is explicitly avoided in the deletion procedure. Since unnecessary internal nodes can only be created by deletion tasks, this directly corresponds to our initial assumption.

The preceding lemma makes clear that there cannot be any degradation in a SkyMap index, no matter what sequence of insertions and deletions is performed. Our next lemma is about index behavior on deletions.

Lemma 3. *Let S be a SkyMap indexing some data set A and $B \subseteq A$. Then, any SkyMap indexing B only has nodes at those positions where there is a node in S.*

Proof. Due to the uniqueness property of SkyMap indexes, any index S' storing only a subset of another index S, can be derived from S by performing a series of individual point deletions and reordering points in leaves. By design of the delete operation, no new nodes are being constructed, but only underflowing nodes are removed. Therefore, S' cannot contain a node at some position where is no node in S.

Corollary 1. *(From Lemma 3) Let A be data set and $B \subseteq A$. Then, the maximum node depth in any SkyMap indexing B is smaller than or equal to the maximum node depth found in any SkyMap indexing A.*

Consequently, the maximum depth of a SkyMap indexing only the skyline of A is at most as large as the maximum depth of any SkyMap indexing the whole data set A.

Finally, let us consider the case of uniformly distributed data, which usually is a good indicator of the general behavior of skyline data structures.

Lemma 4. *Let A be a random set of size n, where each $p \in A$ is a random vector, which has independently and uniformly been drawn from the unit hypercube $[0,1)^d$. Then, the maximum depth of any SkyMap indexing A is $O(\log n)$ in expectation.*

Proof. (Sketch) Since each point p has been drawn uniformly from $[0,1)^d$, the bits of p's Z-address are independent random variates with equal probabilities for each of the outcomes 0 and 1. Therefore, all points spread uniformly over all possible Z-address prefixes, thus implying logarithmic depth.

Corollary 2. *(From Lemma 4 and Corollary 1) The SkyMap index storing the skyline of a random uniformly distributed data set A of size n has maximum node depth $O(\log n)$ in expectation.*

We conclude that SkyMap indexes provide the balancedness needed for quick data access combined with robustness against degeneration caused by frequent updates.

4.5 Skyline Algorithms

To enable the actual processing of skyline queries, we still need an effective way to perform dominance tests on the index. Given a data point p, our IsDominated operation checks whether p is dominated by some point stored in the SkyMap. The algorithm (cf.

```
function IsDominated(p)
    if root = null then return false
    else
        z ← p's Z-address
        return IsDominated (p, z, root, 0, {1, ..., d})

function IsDominated(p, z, node, depth, EQ)
    if node is an internal node then
        (zb_0, ..., zb_{b-1}) ← the first b bits of z
        z' ← z without its first b bits
        for each i = 2^b − 1, ..., 0 with node.children[i] ≠ null do
            (ib_0, ..., ib_{b-1}) ← i in b-bit representation
            EQ' ← EQ
            for each j = depth, ..., depth + b − 1 do
                dim ← (j mod d) + 1
                if dim ∈ EQ then
                    if zb_j < ib_j then EQ' ← EQ' \ {dim}                ▷ p ≺_dim node
                    else if zb_j > ib_j then next i    ▷ p ≻_dim node, continue the outer for loop
            if EQ' = ∅ then return true
            if IsDom.(p, z', node.children[i], depth + b, EQ') then return true
        return false
    else
        for each point q contained in node do
            if p ≺ q then return true
        return false

function Skyline(A)
    Sort A by decreasing Z-addresses; let P be the resulting list of points
    S ← a new empty SkyMap index
    for i = 0, ..., |A| − 1 do
        p ← P[i]
        if no point in S dominates p then Insert p into S
    return the set of points contained in S
```

Fig. 6. Checking whether a point is dominated and computing the skyline of a given data set

Fig. 6) rests on the following observation: When traversing a SkyMap index while looking for points dominating p, one can skip any node (along with all its children) whose corresponding Z-region is worse than p with respect to at least one dimension.

IsDominated works as follows: The SkyMap index is traversed in depth-first search, beginning with those nodes belonging to the largest Z-addresses to visit nodes and points with high domination power first. At each internal node, we scan over all child nodes and compare the length-b Z-address interval used to identify each child to the corresponding interval in p's Z-address. This way, we can easily determine whether p is better in some dimension *dim* than the child node. If this is the case, the child can be immediately be excluded from further traversal. While traversing the index, we continuously maintain a set *EQ* of dimensions, in which the current node *node* and p have found to be equal with respect to the Z-address prefix processed so far. A dimension is eliminated from *EQ* if a child node is known to be better than p with respect to this dimension. During the traversal only those dimensions still contained in *EQ* have to be checked. Whenever we reach a leaf node, p is compared to each of the node's points.

The leaf node capacity c typically will be adapted to the current hardware so that all of the leaf node's points can fit into the CPU cache, which supports extremely fast point comparisons. Moreover, the set *EQ* can be implemented only by performing bitwise operations on ordinary integers. The same is true for all Z-address comparisons. This way, expensive point comparisons again can largely be avoided.

```
function MINSERT(p)
    if some point in the skyline index dominates p then
        Insert p into the data index
    else
        P ← the set of skyline points being dominated by p
        for each q ∈ P do
            Delete q from the skyline index
            Insert q into the data index
        Insert p into the skyline index
```

```
function MDELETE(p)
    if p is not contained in the skyline index then
        Delete p from the data index
    else
        P ← the set of all data points being dominated by p
        Delete p from the skyline index
        for each q ∈ P do
            if no point in the skyline index dominates q then
                Delete q from the data index
                Insert q into the skyline index
```

Fig. 7. Maintaining the skyline in case of changing data

Now, we are able to state our SKYLINE algorithm, which combines the insights of traditional list-based skyline algorithm with the pruning power of SkyMap. It also exploits the following important monotonicity property of Z-addresses:

Lemma 5. *Let $p, q \in [0,1)^d$. If $Z(p) > Z(q)$, then $p \nprec q$.*[1]

Our SKYLINE algorithm works as follows: It first presorts the data set by decreasing Z-addresses, which by the above lemma guarantees that no data point dominates any of its predecessors. Then, an empty SkyMap index is created and the sorted list is scanned linearly. For each point p, the function ISDOMINATED(p) is called. If it returns true, then there already is a point in the index dominating p, thus eliminating p as a skyline point. If the function returns false, p must be a skyline point. In this case, we insert p into the SkyMap index, which always contains the skyline of the data points processed so far. Fig. 6 shows the pseudocode.

For continuously maintaining the skyline of a large database, we propose to use two SkyMap indexes, where the first indexes all current skyline points and the second all remaining database items. To support bulkloading, insertion and deletion, we designed three algorithms: MBULKLOAD, MINSERT, and MDELETE.

Given a data set A, MBULKLOAD creates a valid initial configuration of the two indexes by first computing the skyline of A by means of our SKYLINE algorithm, then removes all skyline points from A, and finally bulkloads A into the second SkyMap index using the BULKLOAD method.

MINSERT and MDELETE require the helper function FINDDOMINATED, which returns a list of all items in a SkyMap index that are dominated by a given point p. The returned list is sorted by decreasing Z-addresses. FINDDOMINATED follows the same approach as our ISDOMINATED function, with only two major differences: First, instead of finishing the traversal at the first dominated point, each the search continues until the whole tree has been traversed; second, all bitwise comparisons operations are performed inverted. As FINDDOMINATED can easily be derived from ISDOMINATED's pseudocode, we abstain from providing an extensive description.

With the help of FINDDOMINATED, maintained insertions and deletions can be performed as shown in Fig. 7. When inserting a new point p, no special action is required if p is dominated, whereas in case p is a new skyline point all current skyline item being dominated by p need to be found and moved to the data index. The opposite happens when deleting an existing point p. Here, no special action is required if p is dominated. Otherwise, all index points being dominated by p need to be found and moved

[1] A proof of this lemma can be found in [9].

to the skyline index if they now become new skyline points. As the result list returned by FINDDOMINATED is ordered by decreasing Z-addresses, we can exploit the same monotonicity property that already proved to be helpful in our SKYLINE method.

We designed our algorithms for in-memory computations. It has been demonstrated in previous work that skylining inherently is CPU-bound and easily become intractable if main memory is scarce (see e.g. [2] or [16]). Fortunately, even the largest data sets so far used in skyline research easily fit into a modern desktop computer's main memory.

5 Evaluation

Beside our own method, we implemented the following state-of-the-art algorithms for skyline computation and maintenance: (1) OSPSPF [16] with the pivoting extension for bulkloading introduced in [7], (2) the ZB-tree [9]. Our experimental setup follows the standard methodology used in skyline research.

Regarding our test data sets, we decided to follow the common methodology in the skylining literature and thus used the three data generators IND (independent data dimensions), CORR (correlated), and ANTI (anti-correlated) as proposed in [2]. Given parameters d and n, these algorithms randomly create data sets having independent, correlated, or anti-correlated dimensions, respectively. In the experiments reported below, *we did not apply any data normalization by means of copulas,* as as found SkyMap's performance on copula-normalized data to be very similar. This indicates that our approach is quite robust with respect to data skewness.

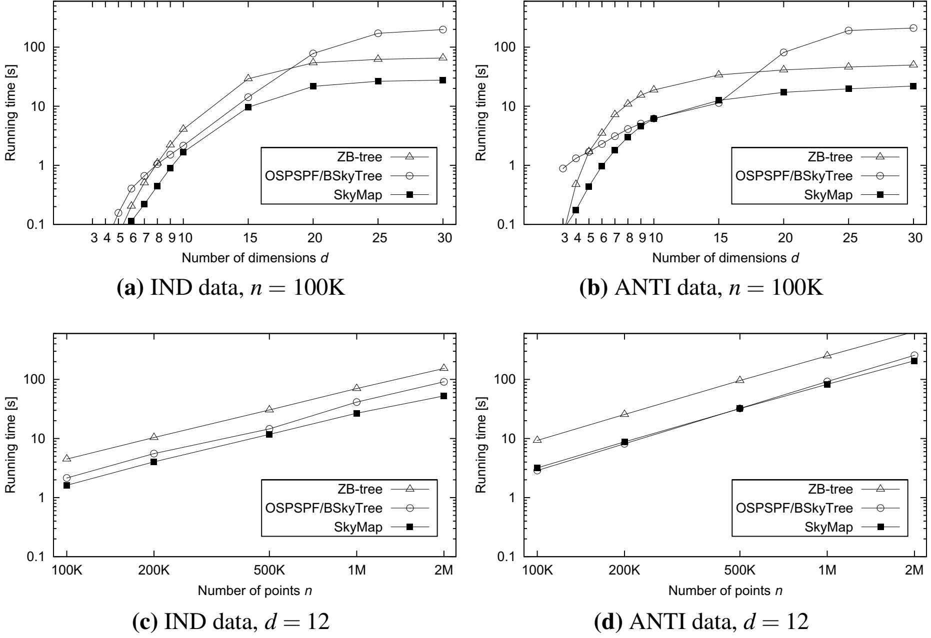
We also wanted to evaluate our method on real-world data sets, but soon realized that all real-world data sets traditionally used in skyline research (e.g., NBA, Corel, Household) are far too small and simplistic to pose a challenge to modern skyline algorithms and thus allow meaningful comparisons among them, a problem that already became apparent in [16], [7], and [9]. To remedy this issue we decided to use a 60-dimensional data set consisting of texture features extracted from $n = 275{,}465$ aerial images, where the skyline consists of 26,817 points. The data set can be downloaded from the web site of the Vision Research Lab at UCSB.[2]

For each of the different choices of d and n we used in our experiments, we randomly generated 10 data sets with each of the three data generators. All running times reported below are averages over the corresponding 10 skyline computations with each algorithm. We did not report any running times below 100 ms due to measurement uncertainty; numbers below this threshold tend to reflect arbitrary delays in memory allocation and data initialization rather than actual performance and scalability.

Our programming language is Java 6. We carefully profiled and optimized all our code to eliminate weak spots. This also included checking all our implementations against existing original code. For example, we compared our version of the partitioning algorithm to the code used in [16], which the authors kindly made available to us.

All experiments have been conducted on a Linux server system equipped with two Intel Core i7 920 2.67 GHz quad-core processors and 20 GB of main memory. However, all our code is single-threaded and uses only a small fraction of the available memory.

[2] `http://vision.ece.ucsb.edu/download.html`



(a) IND data, $n = 100K$ **(b)** ANTI data, $n = 100K$

(c) IND data, $d = 12$ **(d)** ANTI data, $d = 12$

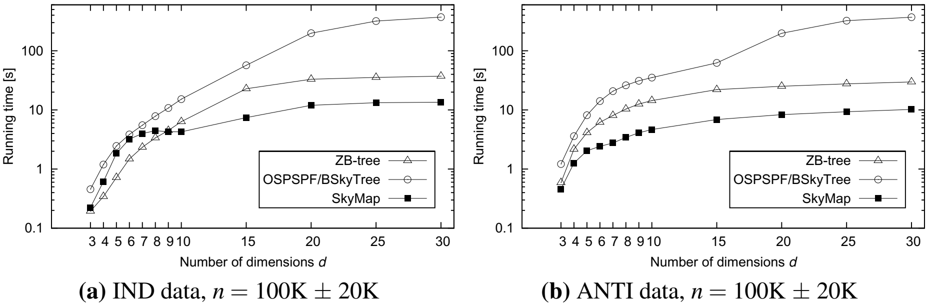
Fig. 8. Skyline queries on synthetic data

Since both the ZB-tree and SkyMap depend on configuration parameters, we tried a variety of different settings and finally ended up with a minimum node size of 20 and a maximum node size of 50 for the ZB-tree; we chose $b = 2$ and $c = 10$ for SkyMap.

5.1 Skyline Computation

We first evaluated our approach in the traditional setting of skyline queries, that is, given a data set A, the task is to compute its skyline. To investigate the influence of data dimensionality, we set $n = 100K$ and varied d over a large range of values. We also evaluated the scalability of our method with respect to n by fixing $d = 12$ and varying n. Fig. 8 depicts our findings. Due to space limitations, we only report performance numbers for IND and ANTI data; results for CORR data are very similar.

As we can see there is no clear winner in the comparison between the ZB-tree and OSPSPF/BSkyTree. The ZB-tree provides better performance for larger d but is worse than OSPSPF/BSkyTree for mid-range values of d. Scalability with respect to n is not an issue for any of the three algorithms. However, in any case, the SkyMap approach significantly outperforms its competitors. Even in the special case $d = 12$, where OSPSPF/BSkyTree's performance comes closest, SkyMap can easily defend its advantage even for different values of n. We also measured scalability in n with respect to many other values of d, but in no case any of ZB-tree or OSPSPF/BSkyTree has been able to perform better than SkyMap. The results on our real-world data set confirm our findings: Computing the skyline takes 13.7 seconds for the ZB-tree, 10.9 seconds for OSPSOF/BSkyTree, and 8.6 seconds for SkyMap.



(a) IND data, $n = 100\text{K} \pm 20\text{K}$ **(b)** ANTI data, $n = 100\text{K} \pm 20\text{K}$

Fig. 9. Skyline maintenance on synthetic data

5.2 Skyline Maintenance

To investigate the performance of our method in the setting of skyline maintenance we used to following three-step task: Given d and n, first n data points are bulkloaded into the database and the skyline is computed. Then, 20% of the data are randomly deleted, where after each deletion the skyline has to be maintained. Finally, a new data set of size 20% is created randomly (according to the original data distribution) and inserted into the database, where again after each single insertion the skyline has to be maintained. Our results for scalability with respect to d as depicted in Fig. 9. We can see that SkyMap consistently performs better than OSPSPF/BSkyTree, sometimes even by two orders of magnitude. Compared to the ZB-tree, the results are twofold. In case of very small skylines, the ZB-tree performs slightly better than SkyMap; however, when it comes to scalability in d and skyline size, SkyMap clearly outperforms the ZB-tree. We also measured scalability in n but always received the same scaling behavior as already depicted in Fig. 8. Therefore, we did not include any further graphics illustrating the fact that no method has scalability issues with respect to n.

5.3 Influence of Parameters

Finally, we investigated the influence of different parameters b and c on the performance of SkyMap indexes. To illustrate all relevant effects, we chose a skyline query example on IND data with $d = 20$. Our results are depicted in Fig. 10. We can see that regardless of the choice of the leaf capacity c, the retrieval performance quickly degrades for

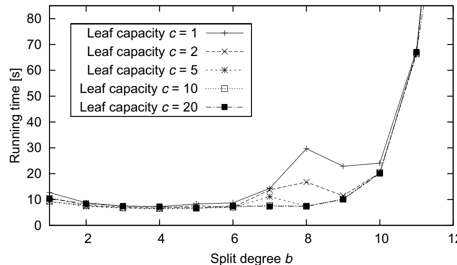


Fig. 10. Influence of parameters b and c

high split degrees b, which also explains the bad performance of OSPSPF/BSkyTree in high-dimensional space we observed previously (recall that quadtrees always split the surrounding space into 2^d partitions). Moreover, our results show performance disadvantages for very small leaf capacities, in particular when b is large. Due to our flexible design, we can always chose a combination of parameters that optimally exploits the specifics of the current hardware. However, our SkyMap approach is robust enough to work well over a wide range of parameters.

6 Conclusion

In this paper, we have shown that the current state of the art in skyling processing is characterized by a tradeoff. One can either have high-performance indexing on static data (OSPSPF/BSkyTree) or high-performance skyline maintenance (ZB-tree), but unfortunately not both. However, easy integration of skyline algorithms into existing database systems calls for a single method that efficiently supports both scenarios. With our SkyMap index we have proposed a solution that successfully resolves this issue, and consistently outperforms previous algorithms on most data sets.

References

1. Bartolini, I., Ciaccia, P., Patella, M.: Efficient sort-based skyline evaluation. ACM Transactions on Database Systems 33(4), 31 (2008)
2. Börzsönyi, S., Kossmann, D., Stocker, K.: The Skyline operator. In: Proceedings of the 17th International Conference on Data Engineering (ICDE 2001), pp. 421–430 (2001)
3. Chan, C.Y., Jagadish, H.V., Tan, K.L., Tung, A.K.H., Zhang, Z.: Finding k-dominant skylines in high-dimensional space. In: Proceedings of the 32th ACM SIGMOD International Conference on Management of Data (SIGMOD 2006), pp. 503–514 (2006)
4. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: Proceedings of the 19th International Conference on Data Engineering (ICDE 2003), pp. 717–719 (2003)
5. Eng, P.K., Ooi, B.C., Tan, K.L.: Indexing for progressive skyline computation. Data and Knowledge Engineering 46(2), 169–201 (2003)
6. Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: An online algorithm for skyline queries. In: VLDB 2002, pp. 275–286 (2002)
7. Lee, J., Hwang, S.: BSkyTree: Scalable skyline computation using a balanced pivot selection. In: Proceedings of the 13th International Conference on Extending Database Technology (EDBT 2010), pp. 195–206 (2010)
8. Lee, J., Hwang, S., Nie, Z., Wen, J.R.: Navigation system for product search. In: Proceedings of the 26th International Conference on Data Engineering (ICDE 2010), pp. 1113–1116 (2010)
9. Lee, K.C.K., Lee, W.C., Zheng, B., Li, H., Tian, Y.: Z-SKY: An efficient skyline query processing framework based on Z-order. The VLDB Journal 19(3), 333–362 (2010)
10. Nelsen, R.B.: An Introduction to Copulas, 2nd edn. Springer, Heidelberg (2006)
11. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. ACM Transactions on Database Systems 30(1), 41–82 (2005)
12. Sagan, H.: Space-Filling Curves. Springer, Heidelberg (1994)
13. Sahni, S.: Tries. In: Mehta, D.P., Sahni, S. (eds.) Handbook of Data Structures and Applications, pp. 28-1–28-20. Chapman and Hall, Boca Raton (2005)

14. Tao, Y., Xiao, X., Pei, J.: Efficient skyline and top-k retrieval in subspaces. IEEE Transactions on Knowledge and Data Engineering 19(8), 1072–1088 (2007)
15. Viappiani, P., Faltings, B., Pu, P.: Preference-based search using example-critiquing with suggestions. Journal of Artificial Intelligence Research 27, 465–503 (2006)
16. Zhang, S., Mamoulis, N., Cheung, D.W.: Scalable skyline computation using object-based space partitioning. In: Proceedings of the 35th ACM SIGMOD International Conference on Management of Data (SIGMOD 2009), pp. 483–494 (2009)