Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Index and retrieve the skyline based on dominance relationship*

Chang XU†, Li-dan SHOU†‡, Gang CHEN, Yun-jun GAO

(*School of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China*)

†E-mail: chinawraith@163.com; should@zju.edu.cn

Received Dec. 17, 2009; Revision accepted Aug. 12, 2010; Crosschecked Dec. 6, 2010

Abstract: In multi-criterion decision making applications, a skyline query narrows the search range, as it returns only the points that are not dominated by others. Unfortunately, in high-dimensional/large-cardinal datasets there exist too many skyline points to offer interesting insights. In this paper, we propose a novel structure, called the dominance tree (Do-Tree), to effectively index and retrieve the skyline. Do-Tree is a straightforward and flexible tree structure, in which skyline points are resident on leaf nodes, while the internal nodes contain the entries that dominate their children. As Do-Tree is built on a dominance relationship, it is suitable for the retrieval of specified skyline via dominance-based predicates customized by users. We discuss the topology of Do-Tree and propose the construction methods. We also present the scan scheme of Do-Tree and some useful queries based on it. Extensive experiments confirm that Do-Tree is an efficient and scalable index structure for the skyline.

Key words: Spatial database, Skyline, Preference queries
doi:10.1631/jzus.C0900003 **Document code:** A **CLC number:** TP391

1 Introduction

1.1 Motivation

Given a d-dimensional dataset and a total order relationship on each dimension, a point p is said to dominate another point q if it is better than or equal to q in all dimensions and better than q in at least one. A skyline is a subset of points in the dataset that are not dominated by any other points. A skyline query is useful in many decision making applications involving high-dimensional datasets (Börzsönyi *et al.*, 2001).

Example 1 Fig. 1 shows a well-known example for the skyline query. A traveler plans to go to a beach. The table lists the information of hotels near the beach. Obviously, a hotel that is cheaper and nearer to the beach is preferred. Consequently, the skyline query returns p_2, p_4, and p_6 as the possible choices.

Unfortunately, in real world applications, the dataset may be large or high-dimensional. In these cases, the skyline query becomes less informative, as it produces too many results. For example, in the 'household' dataset (http://www.ipums.com), which consists of 127 931 six-dimensional data points, a skyline query produces 5774 points.

Some studies have been done on refining the skyline set. In general, they are classified into two categories. The first is ranking the skyline points according to some inherent properties of the data (Zhang *et al.*, 2005; Chan *et al.*, 2006a; 2006b; Lin *et al.*, 2007). As these methods are user-oblivious, they cannot adequately satisfy various user needs. The second combines the skyline and top-k semantics by a user-specific function (Goncalves and Vidal, 2005; Brando *et al.*, 2007; Lee *et al.*, 2007a; Goncalves and

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (Nos. 60803003 and 60970124), the Changjiang Scholars and Innovative Research Grant (No. IRT0652) at Zhejiang University, and the Fundamental Research Funds for the Central Universities (No. 2010QNA5051), China

Vidal, 2009). Nevertheless, defining an appropriate function is unfriendly to normal users. Another common fault of the above methods is that they always need to repeatedly compute the skyline on the original datasets whenever a new query comes. Therefore they are inefficient on high-dimensional/large-cardinal datasets.

id	dist (km)	price (¥)
p1	6	150
p2	2.5	100
p3	1.6	240
p4	1.4	180
p5	1.1	290
p6	0.8	210
p7	1.0	220

Hotels

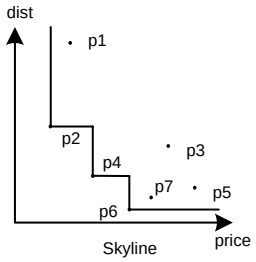

Fig. 1 An example of the skyline

We advocate another approach, which stores the whole skyline set physically and lets the users customize the specific ones themselves. An important fact here is that in skyline-semantic applications, the predicate type in the queries is 'dominance'. Specifically, users always execute 'better' predicates on their preferred dimensions. Therefore, the conventional spatial indices like R-tree (Guttman, 1984) are not suitable here as they are designed originally for region-based queries. We will discuss this in Section 3.3 and empirically show the results in Section 6.

1.2 Do-Tree and customized queries

We aim to design a new type of index, which organizes the skyline based on a dominance relationship. Consider a simple two-dimensional (2D) dataset (Fig. 2a), where the skyline points of the dataset, A, B, C, D, and E, are plotted as black points. In this diagram, F is the point that can just dominate A and B, and G is the point dominating C and D. If we try to combine two neighboring skyline points together and get a parent dominator for each pair recursively, we will obtain a hierarchy of dominators (Fig 2b). We refer to this structure as the dominance tree (Do-Tree).

In the above sample Do-Tree, each skyline point can be located via the dominance relationship. We take point D as an example. Initially we visit the root node I, in which H is the only son who can dominate D. Then we access H, and take G as the next visiting point. Finally we get D on leaf node G.

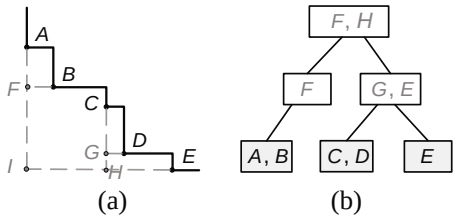

Fig. 2 An example of Do-Tree: (a) skyline; (b) Do-Tree

Intuitively, this is a simple and direct process. On the other hand, since the tree is built on a dominance relationship, the specified queries on the original skyline can always find a reasonable solution.

As will be demonstrated in Section 3.3, in any 2D dataset, we can build a Do-Tree such that, for each skyline point p, there exists only one path from the root to p. When dimensionality is more than 2, however, the situation is complicated. As an example, in a 3D Do-Tree with a node capacity of 2, one must organize the three skyline points: p_1 (1, 1, 3), p_2 (1, 3, 1) and p_3 (9, 2, 2). If we distribute them according to the distance (just as R-tree does), we will place p_1 and p_2 in one node while p_3 in another. Then the parent dominator of p_1 and p_2 is (1, 1, 1), which also dominates p_3. Therefore, when searching p_3, we may also need to visit the node containing p_1 and p_2, which causes an extra access. This indicates that the topology of a Do-Tree should differentiate with the conventional spatial indices, which is extremely suitable for the dominance-based queries.

Once the Do-Tree is built, either skyline point search or dominance-based queries can be conducted on it. We now introduce two useful queries by revisiting Example 1. The formalized definitions of them will be presented in Section 5.

Example 2 In Example 1, if there are many hotels close to the beach, the skyline set retrieved may be large. In this case, a user might impose some constraints on the skylines, e.g., 'the price should be no more than ¥150'. In this way, he/she would obtain p_1 and p_2 as the answer points. In addition, another user, who is a millionare, may want to find the skylines that are 'less than 0.9 km to the beach, whatever the price is'. The only skyline hotel that satisfies the requirement is p_6 in that case.

Example 3 Consider Example 1 again. If a user went to the beach last year and stayed in hotel p_7. This year he/she would like to know whether p_7 is still one of the best choices. Thus, he/she issues a

query to check if p_7 is a skyline. If the answer is 'false', he/she is interested in finding the ones that dominate it. Then the answer skyline hotel is p_6.

1.3 Contributions

To sum up, this paper has the following key contributions:

1. We propose a novel tree structure, namely Do-Tree, to index skyline points. Compared with the conventional skyline refinement approaches, the Do-Tree method supports customized skyline queries based on dominance relationship, so that users can access the skyline according to their preferences.

2. We develop several methods for creating and maintaining the Do-Tree structure.

3. We present several useful dominance-based queries and discuss how to answer them on Do-Tree via the general scan scheme.

4. An extensive experiment evaluation shows that Do-Tree is an efficient index for the skyline on both space consumption and dominance-based skyline retrieval.

2 Related work

Skyline was first introduced into database systems in Börzsönyi *et al.* (2001). Some SQL syntaxes were produced and the criterion of the dataset to the skyline query, specifically correlated, anti-correlated, and independent, was defined. A number of algorithms for computing the skyline were then proposed. Börzsönyi *et al.* (2001) presented two algorithms, block-nested-loops (BNL) and divide-and-conquer (DC). Another variant of BNL is sort-filter-skyline (SFS) (Chomicki *et al.*, 2003), which pre-sorts the dataset according to a monotone function and improves the performance of the BNL algorithm. Tan *et al.* (2001) proposed two new algorithms: Bitmap and Index. Kossmann *et al.* (2002) proposed a nearest neighbor (NN) method to process skyline queries progressively. The branch-and-bound skyline (BBS) (Papadias *et al.*, 2003) is generally considered the best algorithm because it minimizes the I/O cost and visits only the internal or leaf nodes that must be visited. BBS is based on the best-first nearest neighbor algorithm (Hjaltason and Samet, 1999), and it prunes the useless areas using an R-tree. Another excellent algorithm for calculating the skyline was presented in Lee *et al.* (2007b). A close connection between the Z-order curve and skyline processing strategies was observed. A new index structure called ZBtree was proposed to index and store data points based on the Z-order curve. SubSky (Tao *et al.*, 2006; 2007) tries to calculate the skyline using a B+ tree, which is organized by transforming multi-dimensional points to 1D values. SubSky is also applicable in subspace skyline calculating. A novel algorithm SalSa (Bartolini *et al.*, 2008) computes the skyline without scanning the whole dataset, based on either symmetric or asymmetric sorting functions.

In some applications, users may pay attention to only a specified subspace. The concept of SkyCube was first proposed in Yuan *et al.* (2005). Based on some sharing strategies, two novel algorithms, bottom-up and top-down algorithms, were proposed to compute SkyCube efficiently. Pei *et al.* (2005) provided a fundamental framework for multi-dimensional subspace skyline analysis. A simple algorithm, Skyey, was proposed by assembling a data cube algorithm and a sorting-based skyline algorithm. Pei *et al.* (2007) observed a nice relationship between the skyline groups formed by full space skyline objects only and the skyline groups formed by all objects. The relationship is that the former lattice is a quotient lattice of the latter. Then they developed an efficient method, Stellar, to compute the full space skyline only and uses the skyline to shape multi-dimensional skyline groups and their decisive subspaces. The method avoids searching for subspace skylines in all proper subspaces.

When conducting a skyline query in high-dimensional spaces or large datasets, the most troublesome thing is that we always find too many skyline points. A suite of work has been done to refine the skyline according to some inherent properties of the data. Chan *et al.* (2006a) proposed the concept of skyline frequency, which measures the number of subspaces in which a point is a skyline point. They expanded the skyline concept to a wide view called the k-dominant skyline. k-dominant skyline picks the points that are not dominated by any point in any k-dimensional subspace. A top-δ question was also presented in Chan *et al.* (2006a) for a demand of δ points. Three algorithms were proposed to solve the k-dominant skyline problem, which are One-Scan, Two-Scan, and Sorted Retrieval. Lin *et al.* (2007) investigated the problem called 'k representative skyline points' of computing k skyline points

such that the total number of (distinct) data points dominated by the group of the k skyline points is maximized. Top-k dominating (Yiu and Mamoulis, 2007) is a similar idea, but it searches for the points that dominate the most points, which means the top-k dominating query may return some points that are not skylines. Though these methods could refine the skyline points according to the rational rule, they are not flexible enough to satisfy various user needs.

Some other works focused on combining the skyline query and a user-specified ranking function to retrieve the top-k skyline points. Brando *et al.* (2007) ranked the top k tuples in terms of a user-defined score function, while the skyline identifies non-dominated tuples, i.e., tuples for which there does not exist a better one in all user criteria. Goncalves and Vidal (2005) proposed a unified approach that combines paradigms based on ordered scores, and proposed physical operators for SQLf considering skyline and top-k features. Goncalves and Vidal (2009) used discriminatory criteria to induce a total order of the points that comprise the skyline, and recognized the top-k objects based on these criteria. Based on the criteria, an index-based algorithm TKSI was proposed to solve the top-k skyline query. Lee *et al.* (2007a) supported personalized skyline queries as identifying 'truly interesting' objects based on user-specific preference and retrieval size k, after which they solved the problem using a compressed SkyCube.

Our proposal of Do-Tree offers a novel view that indexes the skyline points and grants the refinement to users. Given a Do-Tree, the user could choose a small set of skylines via their dominance-based predicates on specific dimensions. Differentiated from others, the topic of this study focuses on a new style structure and queries based on this.

3 Overview of Do-Tree

This section gives an overview on Do-Tree. We first present some preliminaries and definitions used in this paper. Then we look at the data structures used by Do-Tree. Finally, we define various types of skyline queries supported by Do-Tree.

3.1 Preliminaries and definitions

Given a d-dimensional space $S = \{s_1, s_2, ..., s_d\}$, a set of points $P = \{p_1, p_2, ..., p_n\}$ is said to be a dataset on S if every $p_i \in P$ is a d-dimensional data point on S. We use $p_i.s_j$ to denote the jth dimension value of point p_i and d to denote the number of dimensions of S. The universe on dimension s_i is $[0, U_i]$. For each dimension s_i, we assume there exists a total order relationship, denoted by $\prec$, on its domain values. Here, $\prec$ can be '$<$' or '$>$' relationship according to the user's preference. For simplicity, and without loss of generality, we assume the total order to be '$<$' in the rest of this paper.

The following definitions are used in this paper:

dominate: A point p_i is said to dominate p_j, if and only if $\forall s_k \in S$, $p_i.s_k \leq p_j.s_k$ and $\exists s_t \in S$ so that $p_i.s_t < p_j.s_t$. We use notation $p_i \vdash p_j$ if p_i dominates p_j. Otherwise, we note $p_i \nvdash p_j$.

dominator: A point p is said to be a dominator of a point set $Q = \{q_1, q_2, ..., q_n\}$ if and only if $\forall q_i \in Q$, $p \vdash q_i$. In this case, we note $p \vdash Q$.

min dominator: A point p is said to be the min dominator (MD) of a point set $Q = \{q_1, q_2, ..., q_n\}$ if and only if $p \vdash Q$ and there does not exist another point o that satisfies $o \vdash Q$ and $p \vdash o$. We abbreviate it as $p = \mathrm{MD}(q_1, q_2, ..., q_n)$.

mutex: A point p_i and another point p_j is said to be mutex, if and only if $p_i \nvdash p_j$ and $p_j \nvdash p_i$.

skyline: A point p_i is said to be skyline on S if and only if $\nexists p_j$ in S that $p_j \vdash p_i$. We use $\mathrm{Sky}(P)$ to denote the skyline of P.

dominance area: The dominance area (DA) of a point p is the size of area that p dominates. Formally, $\mathrm{DA}(p) = (U_1 - p.s_1) * (U_2 - p.s_2) * ... * (U_d - p.s_d)$.

3.2 Data structures in Do-Tree

The Do-Tree is a disk-resident tree structure. Each node is stored in one fixed-size page, which is the basic exchange unit between memory and disk. A node N in the Do-Tree contains a set of entries $\{e_1, e_2, ..., e_n\}$ and a min dominator md. Each entry e_i is a tuple like

$$e_i =< \mathrm{key}, \mathrm{id} >,$$

where key is a multi-dimensional point, and id is a pointer to another node or a record. The min dominator md is defined as the min dominator of all the keys of the entries on the same node: md = $\mathrm{MD}\{e_1.\mathrm{key}, e_2.\mathrm{key}, ..., e_n.\mathrm{key}\}$. A leaf node entry stores the spatial attributes of a skyline point and a pointer (record id) to its data record, while an internal node entry stores a child node's md and a

pointer (node id) to it. Just like the B+ tree, all nodes (except the root) in the Do-Tree should be at least half full.

The above definition implies the following lemmas:

Lemma 1 In a Do-Tree, the key of an internal entry dominates all the skyline points that lie in its subtree.

Lemma 2 In a Do-Tree, the key.s_j of an internal entry is determined by the skyline point with the minimum value on s_j in its subtree.

Lemma 3 In a Do-Tree, for each skyline point p on leaf N, p is dominated by the mds of the nodes that reside on the path from the root to N.

3.3 Analysis

While searching for the specific skyline points on Do-Tree, we check the dominance relationship between the predicate and the mds of the nodes to judge the path. If we treat one node access as an I/O cost, the topology structure of the Do-Tree determines the query performance on it. Extra accesses may occur in a Do-Tree with a bad topology. For example, Fig. 3 shows two Do-Trees on a same dataset with five skyline points A, B, C, D, and E. Here Do-Tree I is obviously not a good structure compared to Do-Tree II. As when searching for skyline point B in tree I, we cannot assert which leaf node is the proper path. Therefore, an extra access may occur.

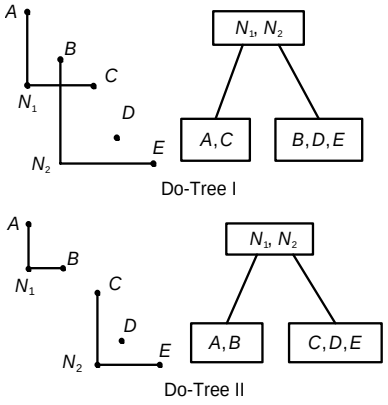

Fig. 3 Two Do-Trees on the same dataset

We will first prove that a 2D Do-Tree can avoid any extra access. Then we will discuss the topology of the Do-Tree in high-dimensional spaces and present the criterion of evaluating it.

Theorem 1 In each 2D dataset, there always exists a perfect Do-Tree. For every skyline point in the tree, there exists only one possible visiting path from the root to it.

Proof Assuming the capacity of the Do-Tree is m and the dimensions are x and y, we create the Do-Tree in a bottom-up way. The entries on the leaf level are the skyline points, and the entries on the upper level are the mds of their son nodes. On each level, sort the entries according to their key.x. Then group the first m entries into the first node, second m entries into the second node, and so on. Repeating this process, we build a Do-Tree. Now we will prove that Do-Tree created in this way is a perfect one.

Between two internal entries e_1 and e_2 at the same level of the above tree, if e_1.key.$x \le e_2$.key.x, then for each skyline point p in the subtree of e_1 and each skyline point q in the subtree of e_2, obviously,

$$p.x < q.x \wedge p.y > q.y \tag{1}$$

holds. The fact that there are two possible paths to a skyline point g means that there exist two entries at the same level both dominating it. Without loss of generality, we assume they are e_1 and e_2, and e_1.key.$x < e_2$.key.x. Assume q is the skyline point in e_2's subtree satisfying $q.x = e_2$.key.x (Lemma 2). Then $q.x \le g.x$. According to Eq. (1), $q.y < e_1$.key.y and e_1.key.$y \le g.y$, so $q.y < g.y$ holds. Thus, we deduce that q dominates g, which contradicts with the assumption that g is a skyline point. Therefore, there is only one path to g. Proof completed.

In a high-dimensional space, it is impossible to produce a perfect Do-Tree in some cases. For example, consider a node capacity of 2 and three 3D points (1, 3, 1), (1, 1, 3), (3, 1, 1). No matter which two points are placed in one node, there will be an md of (1, 1, 1), which dominates the left one. In this case we say an overlap occurs, which may cause extra accesses. A high-level overlap implies more possible extra accesses, because it would produce more possible paths. Theoretically, we have the following quantized formula:

Formula 1 In a Do-Tree with height h and fanout m, for a single skyline point, if there are l leaf nodes dominating it, the possible worst I/O time t of accessing it is within

$$[\frac{l((2/m)^K-1)}{2/m-1}+(h-K), \frac{(m/2)^K-1}{m/2-1}+l(h-K)],$$

where $K=\lfloor \log_{m/2} l \rfloor$.

Formula 1 could be deduced as follows. The possible I/O time when accessing l leaves from the root is decided by the speed of convergency from the bottom to the top. In the best case, the l leaves converge in K levels, with the fan-in $m/2$. Therefore,

$$t=\underbrace{1+1+1+\cdots}_{h-K}+\underbrace{m/2+(m/2)^2+...+l}_{K},$$

which is the left value of Formula 1. In the worst case, K represents the highest level at which the possible paths achieve l. Here,

$$t=\underbrace{1+m/2+(m/2)^2+\cdots+l}_{K}+\underbrace{l+l+...+l}_{h-K}.$$

In a perfect Do-Tree, where l=1, the lower bound of the cost is h. The worst case is when all the leaf nodes dominate the point, where $l=n/(m/2)$, the upper bound of the cost is $l+(l-1)/(m/2-1)$, which equals the size of the tree. According to Lemma 3, clearly in any case a smaller l means fewer overlaps, so that the retrievals are more efficient. We measure the topology structure of Do-Tree via a new definition, the contained coefficient, which is formalized as follows.

Contained coefficient ($\mathcal{C}$): For a constructed Do-Tree, the contained coefficient of a skyline point p is defined as the number of leaves whose mds dominate p. We denote it as $\mathcal{C}(p)$. The contained coefficient of the whole tree is the arithmetic mean of all the skyline points, denoted as $\overline{\mathcal{C}(\mathrm{Sky}(P))}$.

The instruction of building the Do-Tree is to minimize the $\overline{\mathcal{C}(\mathrm{Sky}(P))}$. In a perfect Do-Tree, the $\overline{\mathcal{C}(\mathrm{Sky}(P))}$ is 1, which indicates that the number of I/O times for accessing a skyline point is h.

3.4 Comparison with R-tree

From the analysis in Section 3.3, we conclude that the topology of Do-Tree is different from the conventional spatial indices. We take the most common one, R-tree, as an example. Although R-tree could answer dominance-based queries via the comparison between the predicate and the minimum bounding rectangles (MBRs) (Papadias *et al.*, 2003), it was designed originally for region-overlapped predicates (Guttman, 1984). When building an R-tree, the instruction is minimizing the overlapped area between different nodes. Therefore, the algorithms for redistributing the entries of R-tree always aim at clustering the 'nearer' entries together. When the dimensionality is higher than 2, this will cause a larger $\overline{\mathcal{C}(\mathrm{Sky}(P))}$ in an R-tree. We illustrate this by an example.

Example 4 Assume there is a skyline points set P' including four points: p_1 (1, 1, 3, 5), p_2 (4, 5, 1, 1), p_3 (2, 9, 9, 3), and p_4 (3, 8, 8, 4). The node capacity of a tree is 2. If we organize them using an R-tree, then p_1 and p_2 are placed into one node N_1 and p_3, p_4 into another node N_2 according to the 'nearer' rule. We can compute that their $\overline{\mathcal{C}(P')}$ is 1.5, as p_3 and p_4 may be resident on either N_1 or N_2.

Example 4 shows that the conventional multi-dimensional indices built using the distance-based rule do not fit to the dominance-based queries, which will be empirically demonstrated in Section 6. In fact, if we cluster the four points in Example 4 as (p_1, p_3) and (p_2, p_4), then the $\overline{\mathcal{C}(P')}$ is 1, which means there are no potentially extra accesses at all. This also implies that the splitting strategy of Do-Tree is important to the topology, which will be described in Section 4.3.

Additionally, Do-Tree could gain more space efficiency than R-tree, as it need not store the information about the MBR in the internal nodes. Instead, it maintains only the minimum information for the dominance relationship.

4 Creation of Do-Tree

Section 3.3 proposes a method for creating a perfect Do-Tree in 2D space. However, the method is not scalable when the dimensionality is three or more. As the topological structure is various with the conventional spatial indices, the populating methods for R-tree (Kamel and Faloutsos, 1993; Leutenegger *et al.*, 1997) are not suitable for the Do-Tree either. In general, we build the Do-Tree by inserting the point one by one. To minimize the $\overline{\mathcal{C}(\mathrm{Sky}(P))}$, we choose the most appropriate leaf node for the point and design the strategy when a node splits.

4.1 Insertion on Do-Tree

We describe the common process of inserting a point p into a Do-Tree as follows:

Step 1: Traverse the tree to find a most appropriate leaf N to accommodate p.

Step 2: Insert p into N, if md of N is enlarged, and then update the entry on the parent node. This

process will be repeated until no enlargement occurs or the traversal reaches the root node.

Step 3: If N overflows, splits N. Add the entry of the new node N' into the parent node. This process will be repeated until no overflow/update occurs or the traversal reaches the root node.

In step 1, we find the most appropriate leaf node by iterating from the root to the leaves. In each iteration, we use two criteria to choose the child path. The first criterion is the enlargement of the dominance area. We try to minimize the enlargement of the entry during the insertion, because a larger dominance area increases the possibility of overlaps. If the md of an entry dominates the inserted point, the enlargement is zero, and thus it is the best choice. The second criterion is the number of the entries. If there is a tie for the first criterion, we choose the node with a smallest number of entries to reduce the possibility of splitting.

In step 2, the insertion of p may enlarge the md value of N. In these cases we need to update the entry about N on its parent to keep consistency. This process will be repeated until no further enlargement occurs or the structure modification reaches the root node.

If N overflows after the insertion, we split N into two nodes and distribute the entries on it averagely. The corresponding entry of the new node N' needs to be inserted into the parent node, and it may cause the parent node to overflow. This process will also be repeated. The strategy for splitting the nodes will be described in Section 4.3.

4.2 Creation of Do-Tree from the raw dataset

There are two basic methods for creating a Do-Tree from a dataset. The first method, named direct creation, pre-computes the skyline first, and then inserts them into an empty Do-Tree, via the insertion routine.

We provide another method to create the Do-Tree from the original dataset. In incremental creation, a point will be inserted into the Do-Tree only when there are no points in the tree dominating it. Meanwhile, all the points in the tree it dominates need to be removed from the tree. The process is similar to that of the BNL algorithm (Börzsönyi *et al.*, 2001), but instead of a window, a Do-Tree is maintained during the process and finally completed.

According to Lemma 1, if an internal entry is dominated by the candidate point, all the points in this subtree are excluded from the skyline. Thus, the subtree could be removed directly. Lemma 2 tells us if an internal entry and the candidate point are mutex, then none of the points in its subtree could dominate the candidate. Thus, the subtree need not be accessed anymore.

The process of incremental creation needs some modification on the insert routine. In finding the most appropriate leaf, only the entries that dominate p need to be visited and all the entries dominated by p are pruned immediately. When a leaf entry dominates p, the traversal stops going to the next point. If p survives after the traversal, it is inserted into the Do-Tree as in direct creation. After all the points are processed, the tree needs to be modified to accord with the half full rule. The modification traverses the tree again and distributes all the nodes lower than half full into their brothers. The tree created incrementally is slightly different from the tree created directly. This is because the points that have resided in the tree, but not the final skyline, may enlarge the DA of the nodes. Experiments show, however, that this difference is slight with respect to the topology of the tree. New points can be inserted into a completed Do-Tree using the same method. This indicates that the Do-Tree is a maintainable structure in support of the insertion.

4.3 Node splitting

When a node splits, the entries on it are distributed into the two new nodes averagely. For clarity, we use N, N_1, N_2 to denote the original node and the two new nodes, respectively. Node splitting is formulated as follows:

$$\begin{aligned} &N|\{e_1, e_2, ..., e_n, [N.\text{md}]\} \\ &\quad \rightarrow N_1|\{e'_1, e'_2, ..., e'_{n/2}, [N_1.\text{md}]\} \\ &\quad +N_2|\{e''_1, e''_2, ..., e''_{n/2}, [N_2.\text{md}]\}. \end{aligned}$$

In high-dimensional indices, the splitting strategy is important because it determines the topology of the tree. According to Formula 1, the instruction of building a Do-Tree is to minimize the $\overline{\mathcal{C}(\text{Sky}(P))}$. When a node splits, we directly specialize this goal as the following quantification:

Quantification 1 After splitting, the number of points in $\{e'_1, e'_2, ..., e'_{n/2}\}$ dominated by N_2.md should be as small as possible, and vice versa.

In Section 3.3, we have demonstrated in high-dimensional space that, it is unavoidable to produce overlaps during the splitting. It is difficult to utilize Quantification 1 to evaluate the splitting strategy directly. Therefore, we propose a loose criterion as follows:

Criterion 1 After splitting, N_1.md and N_2.md should be mutex, if possible.

If the md of one new node is dominated by another, then all the entries on it are also dominated. Obviously, this is the worst case and should be avoided. We use Criterion 1 as a primary goal for the splitting. Some candidate algorithms for node splitting in Do-Tree are then demonstrated.

4.3.1 Exhaustive algorithm

The most direct way to find the best splitting is checking all the possible splitting ways and finding the best one according to Quantification 1. However, assuming the capacity of a node is n, there are $C_n^{n/2}/2$ possible splitting ways. Obviously, this is an unacceptable method.

4.3.2 Pick seed algorithm

The pick seed algorithm first picks two entries as the group seeds, and then distributes the remaining entries into the two groups. This is similar to the splitting algorithm in the R-tree (Guttman, 1984). As our goal is to minimize the overlap between the new nodes, the algorithm can be described as follows:

Step 1: For the n entries on the node, pick the two with minimal overlap DA. Assign them into two groups.

Step 2: If one group has owned $n/2$ entries, insert the remaining entries into another group. Algorithm ends.

Step 3: Check the remaining entries and pick the one that, if being placed into one group, creates the minimal DA increment. Go to step 2.

The complexity of the pick seed algorithm is $O(n^2)$. A main law is that because of the half full rule, in the last step all the remaining entries should be poured into one node. This always breaks Criterion 1 when the dimensionality is high. Thus, we abandon this method after numerous experiments and propose a simple but efficient one.

4.3.3 Pick dimension algorithm

We know that for each dimension s_i and each node N, there exists at least one entry that determines the md.s_i of N. After N splits, these points determine the md.s_i of the node they are placed on. Thus, if we distribute them on a different node, the two new nodes are mutex and Criterion 1 is guaranteed. Motivated by this, we develop a pick dimension algorithm, which works as follows:

Step 1: For each dimension s_i, calculate the middle value of all the entries. Initialize a count for s_i to zero.

Step 2: For each entry e, if e.key.s_i = MD.s_i, then for each dimension $s_{i'}$ other than s_i, if $e.s_{i'} <$ middle.$s_{i'}$, count.$s_{i'}$++.

Step 3: Pick the dimension s whose count is nearest to $d/2$. For each entry e that satisfies e.key.$s >$middle.s, place e to N; else, place e to N'.

The algorithm evaluates each dimension s by the following heuristic: If splitting by s, what is the distribution of [N.md]? The one that distributes N.md into two new nodes most averagely is chosen. The complexity of the pick dimension algorithm is only $O(n'd)$, where n' is the capacity of the node.

5 Dominance-based queries on Do-Tree

Generally speaking, queries on the Do-Tree always start from the root node and traverse the proper nodes according to the dominance-based predicate specialized by users. The skyline points on leaf nodes under the predicate constitute the answer. The general scheme of the scan is as described in Algorithm 1.

Some interesting and useful queries are also proposed in this work. We shall define them and present the solutions to Do-Tree in the following.

Constrained skyline: The constrained skyline query provides constraint value c_i on some specific dimensions s_i ($i = 1, 2, ..., d$) and requests for all the skyline points p's that satisfy: $\forall s_i, p.s_i < c_i.s_i$.

According to Lemma 2, for any internal entry e, if $\exists s_i, e.\text{key}.s_i > c_i.s_i$, then we can skip scanning the subtree of the entry. A constrained skyline query visits the Do-Tree and discards the entries that break the constraint. The entries on leaf nodes satisfying the constraint constitute the final answer.

Algorithm 1 GeneralScan(root, predicate)

```
1: HeapInit(h), Set result_set empty;
2: HeapInsert(h, root);
3: while HeapIsNotEmpty(h) do
4:    N ← HeapPop(h);
5:    if N is an internal node then
6:       for each entry e on N do
7:          if predicate(e.key) then
8:             HeapInsert(h, e.id);
9:    else
10:      for each entry e on N do
11:         if predicate(e.key) then
12:            e → result_set;
13: HeapDestroy(h);
14: return result_set.
```

The performance of a constrained skyline search is determined by the constraints. With smaller constraints, fewer points are found and fewer nodes are accessed. If $\forall s_i, c_i.s_i = U_i$, then all the nodes of Do-Tree would be accessed and all the points are the constrained skyline.

IsSkyline: The IsSkyline query takes a point p as its input and returns true if p is skyline, or false otherwise.

According to Lemma 3, we can exclude p from the skyline immediately if we find a leaf entry that dominates it. The opposite case is complicated. P is confirmed to be a skyline only after we are sure there is no entry that dominates p. That means a point is confirmed to be a skyline only after all the possible leaf nodes are visited. Therefore, the traversal stops and returns false immediately if we find one entry on a leaf node whose key dominates p. And it returns true only after the traversal ends normally.

GetDominators: The GetDominators query takes a point p as its input, and returns the skyline point set Q that satisfies: $\forall q \in Q, q \vdash p$.

As in the IsSkyline query, we need only to visit the entries whose values dominate p. The difference is that GetDominators does not stop until the traversal ends. During the traversal, all the leaf entries whose keys dominate the input point are saved into the result set. In the end, the result set is the answer.

We can also conveniently expand the IsSkyline query and GetDominators query to any specific subspace S'. The algorithms are similar to the original ones; specifically, the predicates are checked on S'.

SubSkyline: The SubSkyline query gives a specific subspace S' of the universe, and requests for the skyline in S'.

Yuan *et al.* (2005) proved that any subspace skyline must be a skyline in the total space or there exists at least one skyline point that equals it on each dimension of the subspace. Therefore, the SubSkyline query on Do-Tree does not guarantee to return all the skylines in the subspace, but ensures returning at least one among those with the same attributes. We believe this is enough for most applications.

Considering any subspace of the universe, obviously the following lemma holds:

Lemma 4 In a Do-Tree, the key of an internal entry dominates all points in its subtree in any subspaces.

Lemma 4 indicates that a subtree could be discarded if the key of the corresponding entry is dominated by any point in S'. We aim to reduce the number of nodes to be accessed. The scan scheme needs to be modified as follows. Each time an internal entry is inserted into the heap (line 8 in Algorithm 1), the heap will be sorted according to the entries' DA in decreasing order. The entry with the largest DA should be popped and visited earlier. Once a leaf entry e is inserted into the result set, the points in the result set dominated by e should be removed. After the traversal ends, the points in the result set constitute the answer.

6 Experiments

We evaluate the performance of Do-Tree in terms of both the index size and the query efficiency. For comparison, we also organize the skyline in three forms:

A Heap structure which organizes the skyline points in sequential order. To retrieve the specific skyline, we execute a sequential scan on it.

An R-tree index built by inserting the skyline points one by one, on which the splitting strategy is the linear split (Guttman, 1984). To retrieve a specific skyline, we traverse the tree according to the dominance relationship between MBRs, as presented in Papadias *et al.* (2003).

An R-tree index which is populated using the Hilbert curve method (Kamel and Faloutsos, 1993). We denote it as Hilbert-R-tree.

6.1 Experiment setting

We conducted experiments on synthetic datasets with a variable number of dimensions and distributions. Table 1 lists the parameters used for controlling the generation of the synthetic datasets. The Do-Tree was built using the direct creation method as default.

Table 1 Parameters used in the experiments

Parameter	Description	Default value
d	Dimensionality	10
n	Cardinality	100k
Dist	Distribution	Independent
Univ	Universe	10 000
NS	Node size	4 KB

d: number of attributes of the points; n: size of the dataset. NS: node size of all the structures

The datasets included three classical distributions, namely independent, correlated, and anti-correlated (Börzsönyi *et al.*, 2001). In the correlated dataset, all dimensions were positively correlated to each other. As such, there were few skyline points. In contrast, in the anti-correlated dataset, dimensions were negatively correlated, so that all points were skyline in this type of dataset. The points in the independent dataset were produced uniformly. Each dataset we used is denoted by a string 'type-dimensions cardinality', where the type can be 'I' (independent), 'A' (anti-correlated), or 'C' (correlated), and 'dimensions' indicates the number of dimensions of the data. For example, dataset 'C-20d500k' indicates a correlated dataset of 500k points in a 20-dimensional space.

6.2 Space consumption

We evaluated the space consumption of the above structures on various datasets. Fig. 4 demonstrates two facts. First, as the dimensionality or cardinality increased, the four data structures enlarged as there were more skyline points to be stored. Second, on all the datasets, Heap was always the smallest because it contained no extra information except the skyline points. In the three indices, Do-Tree was about 60% and 100% smaller than the Hilbert-R-tree and R-tree, respectively. This is obviously correct because, compared with the MBR, the size of the entry in Do-Tree is smaller.

Then we studied the size of the Do-Tree on various node sizes. Fig. 5 shows that in most datasets, the size of Do-Tree was smaller with a larger node size, except for the I-12d100k dataset and C-20d700k dataset, in which the 8 KB node tree was the smallest. However, the sizes were very similar in the same dataset. We will show that the topologies of the trees with various node sizes were similar in Section 6.4.

6.3 Query performance

To demonstrate the efficiency of Do-Tree on dominance-based queries, we conducted the queries as mentioned in Section 5 on the four structures. The datasets used here are independent and anti-correlated because there are few skylines in the correlated datasets. In all the experiments, the performance is determined by the number of I/Os, as the data structures are stored on the disk initially.

For constrained skyline queries, the performance was correlated to the constraint we used. We quantified it by a parameter 'selectivity', which represents the percentage of the result to the original skyline size. We evaluated the performance of different structures on various dimensionality, cardinality, and selectivity. Figs. 6a and 6b illustrate the performance on various dimensionality. We can see in both independent and anti-correlated datasets that, the performance of R-tree declined quickly as the dimensionality increased. When the number of dimensions was larger than 6, the performance of R-tree was even worse than that of the sequential scan on the heap. This shows that the conventional structure R-tree is not suitable for the dominance-based queries on high-dimensional datasets. Do-Tree showed the best performance and perfect scalability on various dimensionality.

When the cardinality increased, Do-Tree showed better scalability than the other three structures, especially on anti-correlated datasets (Figs. 6c and 6d). The reason is that the number of skyline points increases rapidly when the anti-correlated datasets become larger, so that the other three structures need more node accesses. Figs. 6e and 6f illustrate that the performance of R-tree declined quickly when the constraint became weaker. In fact, when the selectivity was larger than 1, almost all the nodes of the R-tree needed to be accessed. The performance of Heap was constant with a various selectivity because we always needed to scan the whole heap. However, Do-Tree was much better than the other structures

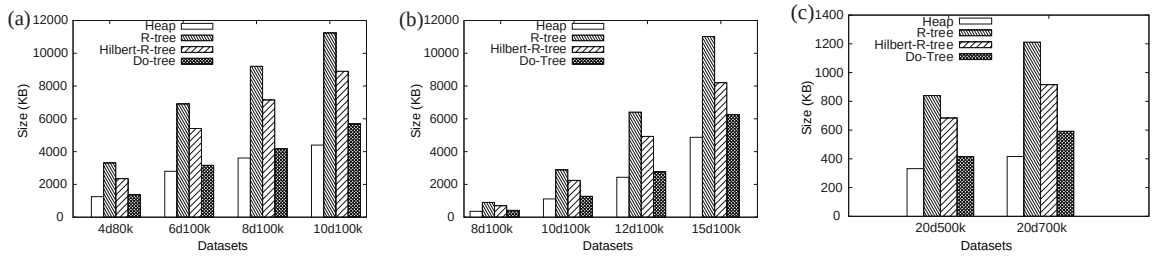

Fig. 4 Space consumption on anti-correlated (a), independent (b), and correlated (c) datasets

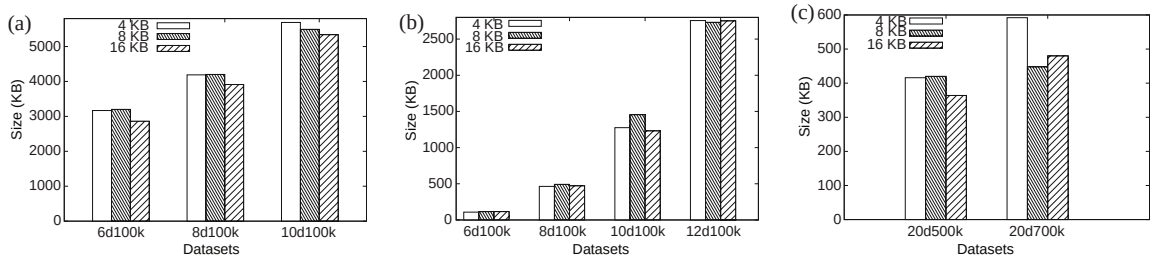

Fig. 5 Space consumption on various node sizes for anti-correlated (a), independent (b), and correlated (c) datasets

under any selectivity.

Fig. 7 illustrates the IsSkyline query on different data structures. In this experiment, we produced 1000 points uniformly as the candidates and checked whether it is skyline in different datasets. Figs. 7a and 7b indicate that, while the performance of other structures deteriorated rapidly when the dimensionality increased, Do-Tree was efficient in answering the IsSkyline query. Fig. 7c indicates the performance of unique skyline queries varying on the dimensionality of subspace. With the decrease of the dimensionality of subspace, the performance of the IsSkyline query improved on all the four structures because the probability that the candidates are dominated by others increased. However, Do-Tree was still more efficient than others when requiring IsSkyline on all subspaces.

Fig. 8 illustrates the GetDominators query on different data structures. The R-tree index showed poor performance on the high-dimensional datasets, as in the constrained query (Figs. 8a and 8b). When the number of the dimensions decreased, the performance of Do-Tree declined (Fig. 8c). The reason is that more points will be returned under lower dimensions, which causes more nodes to be accessed. The performance of the Heap was constant because we always scanned the whole heap to retrieve all the dominators. Fig. 8 demonstrates that Do-Tree was efficient and scalable for the GetDominators query.

For the SubSkyline query, we conducted the query on various dimensionality of subspace on both I-10d100k and A-10d100k datasets. Fig. 9 indicates that the performance of the Heap was consistent on varying subspaces as it needed to scan all the nodes to answer the query. In our experiments, R-tree was unsuitable for searching the subspace skyline in most cases. The performance on Do-Tree was a little higher than that of the Heap when the dimensionality of subspace was near that of the total space. In these cases it pruned few nodes, and almost all the leaves needed to be visited. When the dimensionality was low, Do-Tree was efficient for retrieving skyline in subspaces. Therefore, Do-Tree is suitable for retrieving SubSkyline if the dimensionality of the specified subspace is low, and Heap structure is preferred for high-dimensional subspaces.

The reason why Do-Tree outperforms the conventional spatial indices like R-tree is that its topology is more suitable to the dominance-based queries. To demonstrate this, we picked a subset of skyline points P' randomly and checked the $\overline{\mathcal{C}(P')}$ on both indices. The cardinality of P' was 100. In Do-Tree the leaves that can dominate the skyline points were much less than in R-tree, especially for the anti-correlated datasets (Table 2).

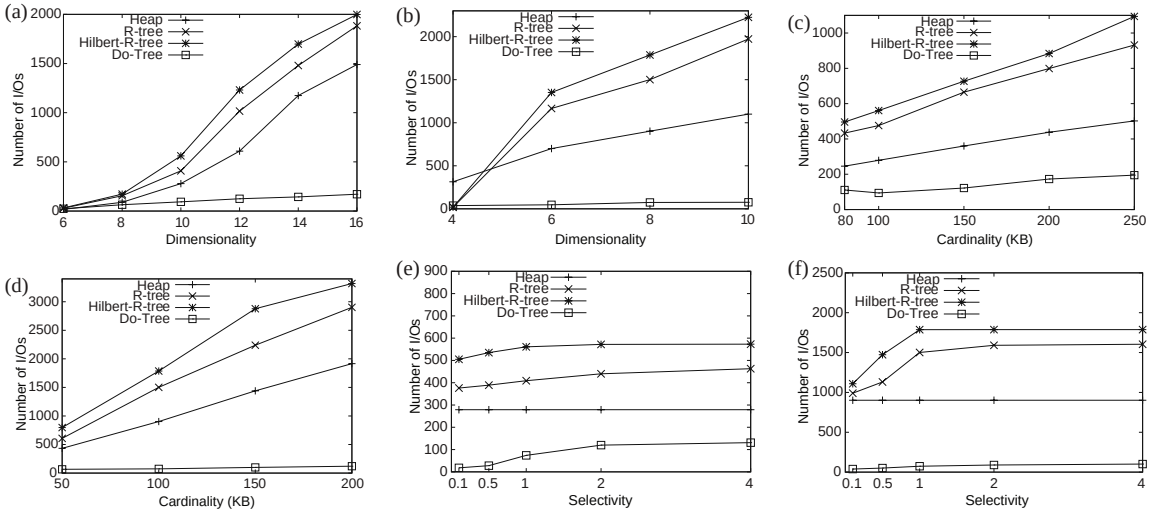

Fig. 6 Performance evaluation of the constrained skyline queries: (a) vs. dimensionality, independent datasets, n=100 KB, selectivity=1; (b) vs. dimensionality, anti-correlated datasets, n=100 KB, selectivity=1; (c) vs. cardinality, independent datasets, d=10, selectivity=1; (d) vs. cardinality, anti-correlated datasets, d=8, selectivity=1; (e) vs. selectivity, independent datasets, d=10, n=100 KB; (f) vs. selectivity, anti-correlated datasets, d=8, n=100 KB

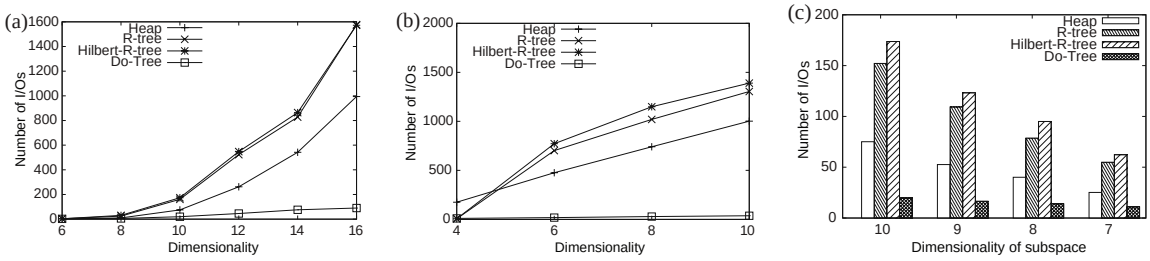

Fig. 7 Performance evaluation of the IsSkyline query on independent datasets with n=100 KB (a), on anti-correlated datasets with n=100 KB (b), and of subspaces for the I-10d100k dataset (c)

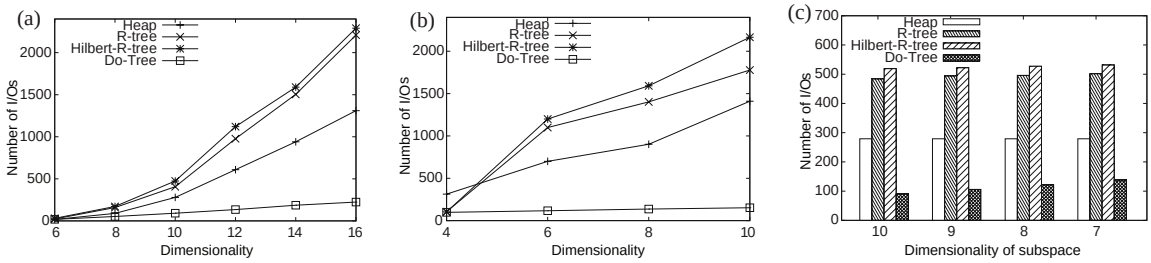

Fig. 8 Performance evaluation of the GetDominators query on independent datasets with n=100 KB (a), on anti-correlated datasets with n=100 KB (b), and of subspaces for the I-10d100k dataset (c)

6.4 Creation of Do-Tree

We first evaluated the performance of creating the Do-Tree using different methods. The number of I/Os increased when the dimensionality became larger (Fig. 10). On the independent datasets, the direct creation method was always more efficient than the incremental creation method. The reason is that the skyline points are only a small part of the original dataset. On the anti-correlated datasets this difference disappeared, as almost all the points were skyline. Therefore, the incremental creation method is preferable for anti-correlated datasets because it reduces the cost of pre-computing the skyline.

Table 2 Contained efficients of the same datasets on different indices

Dataset	$\overline{C(P')}$		
	R-tree	Hilbert-R-tree	Do-Tree
A-6d100k	1134.8	1139.0	26.6
A-8d100k	1515.6	1521.3	24.4
A-10d100k	1585.1	1599.1	23.5
I-8d100k	166.7	181.2	51.1
I-10d100k	507.0	519.9	83.9
I-12d100k	1032.6	1070.1	118.6

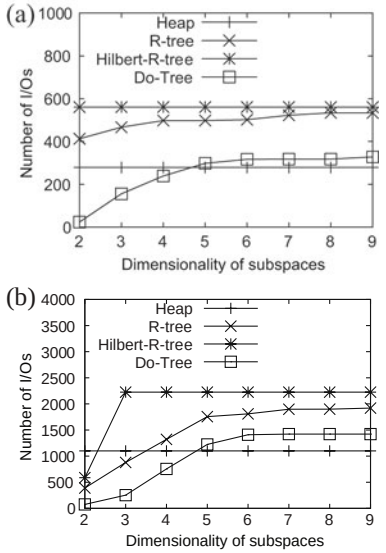

Fig. 9 Performance evaluation of the SubSkyline query on independent datasets, I-10d100k (a) and anti-correlated datasets, A-10d100k (b)

We are also interested in the topology difference between the Do-Tree created using different methods. To evaluate it, we conducted the IsSkyline query on Do-Trees created with different parameters and compared the numbers of I/Os.

The first experiment was conducted on Do-Tree with the same datasets, but using different methods as mentioned in Section 4. In most cases, the Do-Tree created using the incremental method was a little worse than the one created using the direct method (Figs. 11a and 11b). The difference was very small, indicating that their topologies are similar.

Fig. 11c compares the Do-Tree created with different node sizes. We can see that, the situation was different for the three datasets. For the I-10d100k and C-20d700k datasets, the performance on the tree with 4 KB node size was much worse than the other two. For the A-6d100k dataset, the tree with 16 KB node size was more efficient than the other two. This can be explained by the height of Do-Tree. For the

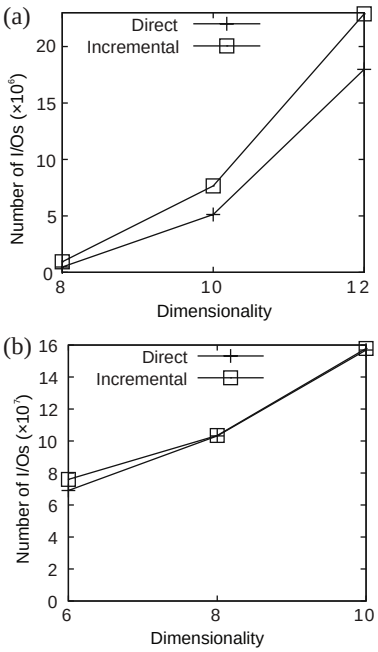

Fig. 10 Performance of creating Do-Tree using direct and incremental creation methods for independent (a) and anti-correlated (b) datasets

A-6d100k dataset, the height of the tree with 16 KB node was 3, while the other two were 2. In the other two datasets, the height of the tree with 4 KB node was 3, while the other two were 2. This indicates that the tree with a lower height always performs better, despite the node size.

To summarize, Do-Tree is an efficient index structure for skyline points on high-dimensional data spaces. It consumes less space than the conventional spatial indices like R-tree. For dominance-based queries, Do-Tree shows good performance and scalability on various dimensionality and cardinality. The topology test indicates that the performance of Do-Tree is sensitive to the height of the tree, but not to the creation method or the node size.

7 Conclusions

Although skyline has become one of the most important operators in multi-dimensional data analysis, there is still no prior work on devising an efficient and flexible structure to manage the skyline points. In this paper we propose the Do-Tree structure to index skyline points in a tree structure based on dominance relationship. We formalize the definition of Do-Tree and present a cost model according to the access type of the tree. We discuss the node

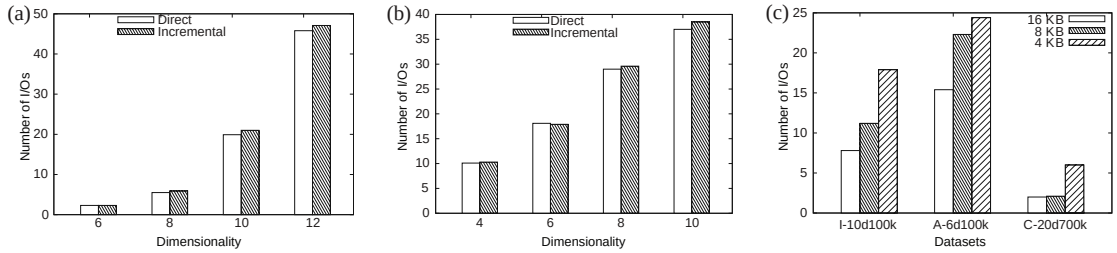

Fig. 11 Performance evaluation of the IsSkyline query on I-10d100k with different creation methods (a), A-10d100k with different creation methods (b), and various node sizes (c)

splitting and propose several algorithms for building Do-Tree from either the skyline set or the original dataset. A suite of dominance-based skyline queries on Do-Tree are discussed as well. Extensive experiments on synthetic datasets show that Do-Tree is an efficient and scalable index structure for skyline.

In the future, we plan to investigate new algorithms for Do-Tree creation. To make Do-Tree more usable, we intend to extend Do-Tree to support deletion and updating actions on the original dataset. Another promising direction is to use Do-Tree to manage skyline in stream data, or in a frequently updated application.

References

Bartolini, I., Ciaccia, P., Patella, M., 2008. Efficient sort-based skyline evaluation. *ACM Trans. Database Syst.*, **33**(4). [doi:10.1145/1412331.1412343]

Börzsönyi, S., Kossmann, D., Stocker, K., 2001. The Skyline Operator. ICDE, p.421-430. [doi:10.1109/ICDE.2001.914855]

Brando, C., Goncalves, M., González, V., 2007. Evaluating Top-k Skyline Queries over Relational Databases. DEXA, p.254-263. [doi:10.1007/978-3-540-74469-6-26]

Chan, C.Y., Jagadish, H.V., Tan, K.L., Tung, A.K.H., Zhang, Z.J., 2006a. On High Dimensional Skylines. EDBT, p.478-495. [doi:10.1007/11687238_30]

Chan, C.Y., Jagadish, H.V., Tan, K.L., Tung, A.K.H., Zhang, Z.J., 2006b. Finding k-Dominant Skylines in High Dimensional Space. SIGMOD, p.503-514. [doi:10.1145/1142473.1142530]

Chomicki, J., Godfrey, P., Gryz, J., Liang, D.M., 2003. Skyline with Presorting. ICDE, p.717-816.

Goncalves, M., Vidal, M.E., 2005. Top-k Skyline: a Unified Approach. OTM Workshops, p.790-799. [doi:10.1007/11575863-99]

Goncalves, M., Vidal, M.E., 2009. Reaching the Top of the Skyline: an Efficient Indexed Algorithm for Top-k Skyline Queries. DEXA, p.471-485. [doi:10.1007/978-3-642-03573-9-41]

Guttman, A., 1984. R-trees: a dynamic index structure for spatial searching. *ACM SIGMOD Rec.*, **14**(2):47-57. [doi:10.1145/971697.602266]

Hjaltason, G.R., Samet, H., 1999. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, **24**(2):265-318. [doi:10.1145/320248.320255]

Kamel, I., Faloutsos, C., 1993. On Packing R-Trees. CIKM, p.490-499. [doi:10.1145/170088.170403]

Kossmann, D., Ramsak, F., Rost, S., 2002. Shooting Stars in the Sky: an Online Algorithm for Skyline Queries. VLDB, p.275-286.

Lee, J.W., You, G.W., Sohn, I.C., Hwang, S.W., Ko, K., Lee, Z., 2007a. Supporting Personalized Top-k Skyline Queries Using Partial Compressed Skycube. WIDM, p.65-72. [doi:10.1145/1316902.1316914]

Lee, K.C.K., Zheng, B.H., Li, H.J., Lee, W.C., 2007b. Approaching the Skyline in Z Order. VLDB, p.279-290.

Leutenegger, S.T., Edgington, J.M., Lopez, M.A., 1997. STR: a Simple and Efficient Algorithm for R-Tree Packing. ICDE, p.497-506. [doi:10.1109/ICDE.1997.582015]

Lin, X.M., Yuan, Y.D., Zhang, Q., Zhang, Y., 2007. Selecting Stars: the k Most Representative Skyline Operator. ICDE, p.86-95. [doi:10.1109/ICDE.2007.367854]

Papadias, D., Tao, Y.F., Fu, G., Seeger, B., 2003. An Optimal and Progressive Algorithm for Skyline Queries. SIGMOD, p.467-478. [doi:10.1145/872757.872814]

Pei, J., Jin, W., Ester, M., Tao, Y.F., 2005. Catching the Best Views of Skyline: a Semantic Approach Based on Decisive Subspaces. VLDB, p.253-264.

Pei, J., Fu, A.W.C., Lin, X.M., Wang, H.X., 2007. Computing Compressed Multidimensional Skyline Cubes Efficiently. ICDE, p.96-105. [doi:10.1109/ICDE.2007.367855]

Tan, K.L., Eng, P.K., Ooi, B.C., 2001. Efficient Progressive Skyline Computation. VLDB, p.301-310.

Tao, Y.F., Xiao, X.K., Pei, J., 2006. Subsky: Efficient Computation of Skylines in Subspaces. ICDE, p.65. [doi:10.1109/ICDE.2006.149]

Tao, Y.F., Xiao, X.K., Pei, J., 2007. Efficient skyline and top-k retrieval in subspaces. *IEEE Trans. Knowl. Data Eng.*, **19**(8):1072-1088. [doi:10.1109/TKDE.2007.1051]

Yiu, M.L., Mamoulis, N., 2007. Efficient Processing of Top-k Dominating Queries on Multi-dimensional Data. VLDB, p.483-494.

Yuan, Y.D., Lin, X.M., Liu, Q., Wang, W., Yu, J.X., Zhang, Q., 2005. Efficient Computation of the Skyline Cube. VLDB, p.241-252.

Zhang, Z.J., Guo, X.Y., Lu, H., Tung, A.K.H., Wang, N., 2005. Discovering Strong Skyline Points in High Dimensional Spaces. CIKM, p.247-248. [doi:10.1145/1099554.1099610]