6-2010

Z-SKY: An Efficient Skyline Query Processing Framework Based on Z-Order

Ken C. K. LEE
Pennsylvania State University

Wang-chien LEE
Pennsylvania State University

Baihua ZHENG
Singapore Management University, bhzheng@smu.edu.sg

Huajing LI
Pennsylvania State University

Yuan TIAN
Pennsylvania State University

Citation

LEE, Ken C. K.; LEE, Wang-chien; ZHENG, Baihua; LI, Huajing; and TIAN, Yuan. Z-SKY: An Efficient Skyline Query Processing Framework Based on Z-Order. (2010). *VLDB Journal*. 19, (3), 333-362.
Available at: https://ink.library.smu.edu.sg/sis_research/1985

Published in VLDB Journal, June 2010, Volume 19, Issue 3, pp. 333-362.
http://dx.doi.org/10.1007/s00778-009-0166-x

Z-SKY: an efficient skyline query processing framework based on Z-order

Ken C. K. Lee · Wang-Chien Lee · Baihua Zheng · Huajing Li · Yuan Tian

Abstract Given a set of data points in a multidimensional space, a skyline query retrieves those data points that are not *dominated* by any other point in the same dataset. Observing that the properties of Z-order space filling curves (or Z-order curves) perfectly match with the dominance relationships among data points in a geometrical data space, we, in this paper, develop and present a novel and efficient processing framework to evaluate skyline queries and their variants, and to support skyline result updates based on Z-order curves. This framework consists of *ZBtree*, i.e., an index structure to organize a source dataset and skyline candidates, and a suite of algorithms, namely, (1) *ZSearch*, which processes skyline queries, (2) *ZInsert*, *ZDelete* and *ZUpdate*, which incrementally maintain skyline results in presence of source dataset updates, (3) *ZBand*, which answers skyband queries, (4) *ZRank*, which returns top-ranked skyline points, (5) *k-ZSearch*, which evaluates k-dominant skyline queries, and (6) *ZSubspace*, which supports skyline queries on a subset of dimensions. While derived upon coherent ideas and concepts, our approaches are shown to outperform the state-of-the-art algorithms that are specialized to address particular skyline problems, especially when a large number of skyline points are resulted, via comprehensive experiments.

K. C. K. Lee (✉) · W.-C. Lee · H. Li · Y. Tian
The Department of Computer Science and Engineering,
The Pennsylvania State University, University Park,
PA 16802, USA
e-mail: cklee@cse.psu.edu

W.-C. Lee
e-mail: wlee@cse.psu.edu

H. Li
e-mail: huali@cse.psu.edu

Y. Tian
e-mail: yxt144@cse.psu.edu

B. Zheng
School of Information Systems,
Singapore Management University, Singapore, Singapore
e-mail: bhzheng@smu.edu.sg

Keywords Skyline query · Skyline query result update · Index · Search algorithm · Z-order space filling curve

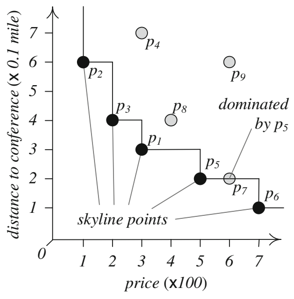
1 Introduction

Given two data points p and p' in a multidimensional space, p is said to dominate p' if p is strictly better than p' on at least one dimension and p is not worse than p' on the other dimensions. To illustrate the idea of dominance relationships, Fig. 1 gives a hotel finding example (i.e., our running example) in which a conference participant looks for a hotel based on two criteria, *price per night* and *distance to the conference venue*. Figure 1a lists 9 hotel records and their values and Fig. 1b depicts the geometrical representation of the hotels in a 2D space with each dimension representing one criterion. Hotels p_4, p_7, p_8 and p_9 are all dominated. For instance, p_7 is dominated by p_5 as it is more expensive than p_5, although both of them are equally good in terms of distance to the conference venue. In database systems, queries specialized to search for those non-dominated data points are named *skyline queries*, and their corresponding results are called *skyline results*. In the example, hotels p_1, p_2, p_3, p_5 and p_6 form a skyline result. Individual data points in a skyline result are *skyline points*.

With a very wide application base in various domains such as multi-preference analysis and decision making, skyline queries have received a lot of attentions [4,5,9,13,16,22,26,29,31,34]. Most of the existing works in the literature proposed and examined different techniques to improve the skyline search performance [4,16,22,26] and to efficiently

Hotel	Price (× 100)	Distance (× 0.1)
p_1	3	3
p_2	1	6
p_3	2	4
p_4	3	7
p_5	5	2
p_6	7	1
p_7	6	2
p_8	4	4
p_9	6	6

(a)



(b)

Fig. 1 An example skyline query (our running example): **a** hotel (price, distance), **b** price, distance space

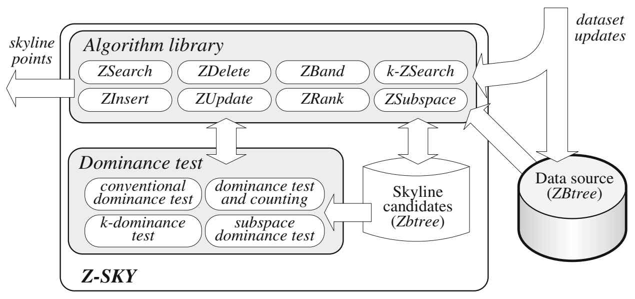


Fig. 2 The Z-SKY skyline query processing framework

update cached skyline results [22,31] in presence of dataset updates. Besides, skyline query variants, such as *skyband* query [22], *top-ranked* skyline query, *k-dominant* skyline query [5] and *subspace* skyline query [9,29] have been recently derived. A skyband query returns data points that are not dominated by more than b other data points (where b specifies the width of a skyband query result interested by users). A top-ranked skyline query finds t skyline points, each of which dominates the most data points, where t is a query parameter. A k-dominant skyline query retrieves a representative subset of skyline points from a high-dimensional data set by considering dominant relationships on *arbitrary* k dimensions rather than all d dimensions (where $k \leq d$). Last but not least, a subspace skyline query performs a skyline search on a specified subset of dimensions. We will discuss them in details in the next section.

Skyline query processing and skyline result updates are expensive operations. Their costs are mainly constituted by I/O costs in accessing data from a tertiary storage (e.g., disks) and CPU costs spent on dominance tests. In particular, the CPU costs can be very high due to exhaustive data point comparisons. Hence, *search efficiency* and *update efficiency* are the two most important performance criteria to skyline query processing and skyline result maintenance. In addition, as skyline queries have been considered as an analytical tool in some commercial database systems [6], it is definitely important to develop a skyline query processing framework that can be easily extended to deal with different skyline query variants. Although various techniques have been recently proposed to address different but closely related research issues raised in skyline queries, almost all of them are developed independently and lack of provisioning a generic framework to tackle all those issues in a seamlessly integrated fashion. Therefore, techniques tailored to facilitate some particular skyline query variants would introduce performance burdens to others and/or skyline result updates, if a framework needs to support skyline queries, skyline result updates and skyline query variants, simultaneously.

Based on our preliminary studies [17], accessing and organizing data points based on a Z-order space filling curve (or Z-order curve, in short) can improve the skyline query evaluation and skyline result update performance, and can support skyline query variants, simultaneously, because of the following three reasons:

1. Z-order curve provides the monotonic access order of data points. This access order can help identifying data points, which are likely to be a part of skyline result and that can dominate many data points, early. Moreover, this avoids candidate reexaminations that removes false hits, and provides progressive result delivery.
2. Z-order curve clusters data points based on their locations in a multidimensional space. With data points clustered and represented as *blocks*, the dominance tests between groups of data points in terms of blocks are accelerated. This *true* block-to-block dominance test cannot be achieved by any existing work. Search space pruning is also facilitated by such block-based dominance tests.
3. Z-order curve is one of the widely adopted dimension reduction techniques [11,24]. Skyline algorithms developed based on Z-order curves are therefore inherently scalable to high data dimensionality.

Motivated by the demand of an efficient and extensible framework for skyline queries and inspired by an observed strong connection between Z-order curves and the dominance relationships among data points in a geometrical data space, we develop and present a novel skyline query processing framework named *Z-SKY* for skyline queries and variants, and skyline result maintenance based on Z-order curves, in this paper. To the best of our knowledge, this is the first work reported in the literature that addresses both the efficiency of processing skyline queries, especially for large datasets with high data dimensionality, and the extensibility of a processing framework to support skyline query variants.

The high-level view of our Z-SKY framework is depicted in Fig. 2. It consists of four main components, namely, (i) a data source ($\mathcal{SRC}$), (ii) a set of skyline candidates ($\mathcal{SL}$), (iii) an algorithm library, and (iv) dominance tests.

Specifically, $\mathcal{SRC}$ is a set of source data points indexed by a ZBtree. ZBtree indexes data points based on their values on a Z-order curve. Second, $\mathcal{SL}$ maintains skyline candidates indexed by another ZBtree. Third, the dominance tests provide different types of dominance relationship tests, such as (conventional) dominance tests, dominance and counting that determines the number of dominating points for a data point, k-dominance tests and subspace dominance tests that support k-dominant skyline queries and subspace skyline queries, respectively. Fourth, the algorithm library maintains a suite of algorithms, namely, (1) *ZSearch*, which processes skyline queries, (2) *ZInsert*, *ZDelete* and *ZUpdate*, which incrementally update skyline query results, (3) *ZBand*, which evaluates skyband queries, (4) *ZRank*, which returns skyline points that dominate the most data points, (5) *k-ZSearch*, which answers k-dominant skyline queries and (6) *ZSubspace*, which performs skyline searches on specified subsets of dimensions. Upon receiving a skyline query (or its variant), a corresponding algorithm is then invoked to access and examine data points from $\mathcal{SRC}$ with corresponding dominance tests. The candidates are kept in $\mathcal{SL}$. After evaluations, skyline points are delivered. Likewise, to maintain a skyline result in presence of dataset updates, a skyline result update algorithm is triggered that determines the change of a result preserved in $\mathcal{SL}$ and accesses required data points from $\mathcal{SRC}$ if needed. It is noteworthy that the underlying operations and data structures follow a coherent idea and concept developed based on Z-order curves. Finally, to evaluate our Z-SKY framework, we conducted a comprehensive performance evaluation and compared our approaches with the state-of-the-art algorithms.

The rest of this paper is structured as follows. In Sect. 2, we discuss the skyline query properties and review existing works. In Sect. 3, we analyze why Z-order curve perfectly matches with dominance relationships among data points that in turn leads to the introduction of *ZBtree* and discuss its manipulations. Section 4 details the *ZSearch* algorithm to evaluate skyline searches. Section 5 presents the *ZInsert*, *ZDelete* and *ZUpdate* algorithms to handle skyline result updates. Section 6 discusses the *ZBand*, *ZRank*, *k-ZSearch* and *ZSubspace* algorithms to support skyline query variants, namely, skyband queries, top-ranked skyline queries, k-dominance skyline queries and subspace skyline queries, respectively. Section 7 evaluates the performance of our proposed algorithms in comparison with the start-of-the-art approaches. Finally, Sect. 8 concludes the paper.

2 Preliminaries

In this section, we formulate the skyline queries and identify their properties and then define those skyline query variants to be addressed in this paper, followed by a review of some related works.

2.1 Definitions and properties

Given a d-dimensional space $S = \{s_1, s_2, \ldots, s_d\}$ that covers a set of data points $P = \{p_1, p_2, \ldots, p_n\}$ (such that every p_i is a data point in S), the dominance relationship between data points and the skyline queries are defined in Definitions 1 and 2, respectively. We use $p_i.s_j$ to denote the jth dimensional value of p_i, and assume the existence of a total order relationship, either '$<$' or '$>$', on each dimension. Without loss of generality, we consider that $p \cdot s_j$ is better than $p' \cdot s_j$ if $p \cdot s_j < p' \cdot s_j$, throughout the paper.

Definition 1 Dominance. Given $p,\ p' \in P$, p dominates p' (denoted by $p \vdash p'$) iff $\forall s_i \in S$, $p{\cdot}s_i \leq p'{\cdot}s_i \wedge \exists s_j \in S$, $p{\cdot}s_j < p' \cdot s_j$; otherwise, p does not dominate p' (denoted by $p \not\vdash p'$).

Definition 2 Skyline query. A skyline query retrieves those data points in P that are not dominated by any other point. We denote a skyline result by U, and formally, $U = \{p \in P \mid \ \nexists p' \in P - \{p\},\ p' \vdash p\}$.

We observe two skyline query properties, namely, *transitivity* and *incomparability*, which provide important insights to facilitate the development of our index and algorithms, as stated below.

Property 1 Transitivity. *Given $p, p', p'' \in P$, if p dominates p' and p' dominates p'', then p dominates p'', i.e., $p \vdash p' \wedge p' \vdash p'' \Rightarrow p \vdash p''$.*

Property 2 Incomparability. *Given $p,\ p' \in P$, p and p' are incomparable if they do not dominate each other, i.e., $p \not\vdash p' \wedge p' \not\vdash p$.*

Based on the transitivity property, if dominating points are always processed before their dominated data points, any data point, which passes the dominance tests against all the skyline points obtained ahead of it, is guaranteed to be a skyline point. This property inspires the sorting-based approaches [8,13]. On the other hand, if two data points are known to be incomparable in advance, dominance tests between them can be avoided. This incomparability inspires the divide-and-conquer approaches [4].

Generally speaking, a skyline query classifies data points into a set of skyline points (U) and a set of non-skyline points (P-U) and returns U. Some non-skyline points may be dominated by only a few data points and they are incomparable to all the remaining skyline points. Revisit our hotel example in Fig. 1. Hotel p_7 is only dominated by hotel p_5. In other words, p_7 is not dominated by all the other skyline points, namely, p_1, p_2, p_3 and p_6. In this case, if the user is interested in p_5 but it is not available, p_7 could be the next candidate for her to consider. To retrieve data points that are not dominated by more than a certain number of data points, the *skyband queries* are defined in Definition 3.

Definition 3 Skyband query. Let D_p represent the set of data points that dominate p, i.e., $D_p = \{p' \in P - \{p\} \bigm| p' \vdash p\}$. Given an integer b (where $b \geq 1$), a skyband query retrieves all the points p which are dominated by b or less other points. We denote the skyband result of the query by U_b and $U_b = \{p \in P \bigm| |D_p| \leq b\}$.

Besides, the conventional skyline query ignores a fact that skyline points might have different number of dominated data points. In many cases, skyline points that have smaller number of dominated data points are those which have a few better attributes and have many less competitive attributes. In our hotel example, p_6 has no dominated point at all because of its extremely high price though being the closest to the conference. Referring one's number of dominated points to as its dominating power, we consider those skyline points with the greatest dominating power (with the most dominated data points) to be the most representative. Then, given a desired number of returned top-ranked skyline points, t, the *top-ranked skyline* queries are defined in Definition 4 to retrieve t skyline points that dominate the most data points.

Definition 4 Top-ranked skyline query. Let E_p represent a set of data points that are dominated by p, i.e., $E_p = \{p' \in P \mid p \vdash p'\}$. A point p is said to have greater dominating power than p' if $|E_p| > |E_{p'}|$. Given an integer t, a top-ranked skyline query retrieves t skyline points from U that have the largest dominating power. We denote the top-ranked skyline result of the query by U_t such that $U_t \subseteq U$, $|U_t| \leq t$ and $\forall p \in U_t, \forall p' \in (U - U_t), |E_p| > |E_{p'}|$.[1]

Another variant is the k-dominant skyline queries that filter out skyline points with only a few good attributes. Instead of all the d dimensions, a k-dominance relationship considers arbitrary k out of all the d dimensions, as defined in Definition 5. With a small k, the size of a skyline result can be reduced [5]. The definition of k-dominant skyline queries is provided in Definition 6.

Definition 5 k-dominance. Given $p, p' \in P$, p k-dominates p' (denoted by $p \vdash_k p'$) iff $\exists S' \subseteq S, |S'| = k, \forall s_i \in S'$, $p.s_i \leq p'.s_i \wedge \exists s_j \in S', p.s_j < p'.s_j$; otherwise, p does not k-dominate p' (denoted by $p \not\vdash_k p'$).

Definition 6 k-dominant skyline query. A k-dominant skyline query retrieves a subset of data points in P that are not k-dominated by any other point. We denote a k-dominated skyline result of the query by U_k and U_k is $\{p \in P \mid \forall p' \in P - \{p\}, p' \not\vdash_k p\}$.

[1] If $|U| < t$, we collect U_t as the entire U.

According to the k-dominance relationship, skyline points that are *only* good at a few dimensions are very likely to be k-dominated by others and hence the k-dominant skyline result size is smaller than a conventional skyline result. It is important to note that data points may get dominated by other data points sorted behind them according to any monotonic access order. This is because a data point p k-dominating another point p' can also get k-dominated by p' at the same time, but in different k dimensions. This is so-called *cyclic dominance* [5]. In this case, p and p' are not k-dominant skyline points. Thus, candidate reexaminations are needed which incurs larger processing overhead. Further, determining whether p k-dominates p' requires to compare at least k dimensional values. Meanwhile, examining whether p does not k-dominate p' needs to examine no less than $(d - k + 1)$ dimensional values.[2] Therefore, each dominance test based on the k-dominance relationship incurs a larger processing overhead than the conventional dominance test.

Different from k-dominant skyline queries, subspace skyline queries retrieve data points that are not dominated by others based on a specified subset of d dimensions (or called a subspace). Accordingly, Definition 7 defines a subspace-dominance relationship between points and Definition 8 formalizes these subspace skyline queries.

Definition 7 Subspace dominance. Given $p, p' \in P$ on a subspace S' ($\subset S$), p S'-dominates p' (denoted by $p \vdash_{S'} p'$) iff $\forall s_i \in S', p.s_i \leq p'.s_i \wedge \exists s_j \in S', p.s_j < p'.s_j$; otherwise, p does not S'-dominate p' (denoted by $p \not\vdash_{S'} p'$).

Definition 8 Subspace skyline query. A subspace skyline query retrieves a subset of data points in P that are not S'-dominated by any other points. We denote the subspace skyline result of the query by U_s and $U_s = \{p \in P \mid \forall p' \in P - \{p\}, p' \not\vdash_{S'} p\}$.

The evaluation of subspace skyline queries would be intuitively treated as conventional skyline queries, except only certain dimensions are considered. However, it is not trivial to have a single data organization to support subspace skyline queries on all possible subsets of dimensions, simultaneously. Recall our example. A subspace skyline query on the price dimension can be facilitated if data points are arranged in an ascending price order, i.e., p_2, p_3, p_4, etc. Likewise, another subspace skyline query for the distance dimension prefers objects sorted according to their distances. Clearly, these orders are different and not favorable to conventional skyline queries that consider both the dimensions.

[2] Among the k dimensions, p has at least one dimensional value better than p' and the rest of p is not worse than that of p'. Meanwhile, as p has $d - k + 1$ dimensional values worse than p', p does not k-dominates p'.

Table 1 Representative skyline query algorithms

	BNL, Bitmap	D&C	SFS, SaLSa, LESS	Index, NN, BBS
D&C	X	✓	X	✓
Sorting	X	X	✓	✓

2.2 Related work

Next, we review representative algorithms for skyline queries and skyline updates as well as skyline variants, such as skyband queries, top-ranked skyline queries, k-dominant skyline queries and subspace skyline queries.

2.2.1 Skyline query processing

Existing skyline query processing algorithms that include Block-Nested Loop (BNL) [4], Bitmap [26], Divide-and-Conquer (D&C) [4], Sort-Filter-Skyline (SFS) [8], SaLSa [2], LESS [13], Index [26], Nearest Neighbor (NN) [16], and Branch and Bound Search (BBS) [22], can be roughly classified based on the adoption of two popular techniques, namely, *divide-and-conquer* (D&C) and/or *sorting*. Table 1 gives a short summary. BNL and Bitmap are brute force algorithms. In the following, we review representative divide-and-conquer, sorting, and hybrid algorithms.

D&C algorithms. D&C divides a dataset into several partitions small enough to be loaded into the main memory for processing [4]. A local skyline for each partition is computed. The global skyline is obtained by merging all local skylines and removing dominated data points.

Sorting-based algorithms. SFS [8] is devised based on an observation that by having a dataset presorted according to a certain monotone scoring function such as the entropy, sum or minimum of attribute values, it is guaranteed that data points must not be dominated by others sorted behind them and the partial query result can be delivered immediately. SFS sequentially scans the sorted dataset and keeps a set of skyline candidates. Those not dominated by skyline candidates should be a part of the skyline. SaLSa [2] presorts a dataset based on the records' minimum attribute values and it improves SFS by terminating a search whenever an examinee data point whose minimum attribute value is greater than the MiniMax (i.e., the minimum among the maximum) of examined data points' attribute values is reached. This termination condition guarantees that all later examined data points should not be skyline points; and this early termination avoids scanning the entire dataset. However, SFS and SaLSa have no or limited search space pruning capability and inevitably examine and compare all the individual data points in a pairwise fashion. Also, dominance tests in SFS and SaLSa are based on an *exhaustive scan* on existing skyline candidates. Unless the number of skyline points is very small, they tend

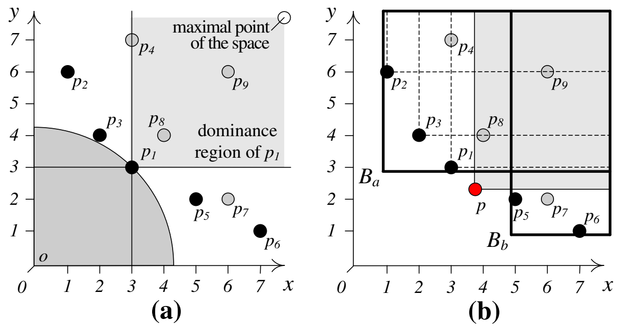


Fig. 3 BBS and main memory R-tree. **a** BBS, **b** main memory R-tree

to be CPU-bound. LESS [13] combines external sort and skyline search in a multi-pass fashion. It reduces the sorting cost and provides an attractive asymptotic best-case and average-case performance based on an assumption that the number of skyline points is small.

Hybrid algorithms. Index [26], NN [16] and BBS [22] are hybrid approaches that use both divide-and-conquer and sorting techniques in skyline query processing. Here, we review BBS, i.e., currently the most efficient online skyline search algorithm. BBS is based on iterative nearest neighbor search [14]. In Fig. 3a that illustrates BBS with the nine hotel example points, p_1, the nearest neighbor (nn) to the origin in the whole space, is identified as the first skyline point since the empty vicinity circle that centers at the origin and touches p_1 ensures that there is no data points dominating p_1. Then, data points (i.e., p_4, p_8 and p_9) fallen into the *dominance region* of p_1, i.e., the region bounded by p_1 and the maximal point of the entire space, are not skyline points and can be safely discarded from a detailed examination. Based on the same idea, the second nn to the origin, p_3, not dominated by p_1, is another skyline point. Next, p_5, the third nn, is retrieved and its dominated point p_7 is removed. Finally p_2 and p_6 are retrieved and the search terminates.

BBS adopts one R-tree to index the source dataset because R-tree can facilitate iterative nn search. It also uses a *heap* to keeps track of unexamined index nodes and data points in non-decreasing distance order with respect to the origin of the space so that repeated accesses of R-tree nodes can be alleviated, and a *main memory R-tree* to index the dominance regions of all skyline points. Via an additional main memory R-tree, BBS performs dominance tests on every examinee (i.e., data point or index node) by issuing an enclosure query. If an examinee is entirely enclosed by any skyline candidate's dominance region, it is dominated. In Fig. 3b, p_8, an examinee data point, is compared with B_a and B_b, the minimum bounding boxes of two leaf nodes in the main memory R-tree. As it is in B_a, not B_b, p_8 is possibly dominated by some data points enclosed by B_a but certainly not B_b. Next, p_8 is compared with the dominance regions of all the data points inside B_a and found to be dominated by p_1 (and p_3).

However, as data dimensionality increases, the performance of R-trees and main-memory R-trees that BBS depends on deteriorates (due to curse of dimensionality [3]). Even worse, some data points (or index nodes) that are far away from the origin are loaded into the heap as their enclosing nodes (or parent nodes, respectively) appeared the closest to the origin. As a result, BBS, sometimes, loads some entries earlier than they are needed. Thus it imposes a high contention on the runtime memory. Besides, dominance tests based on main-memory R-trees are actually not efficient because the examinee data points or index nodes have to reach the bottom level to determine by what skyline candidates it is dominated. In other words, dominance tests are performed based on point-to-block comparison, which is clearly less efficient than block-to-block comparison as provided by our proposed algorithms.

Our ZSearch algorithm (to be presented in Sect. 4) is also a hybrid approach. It processes a skyline query by accessing data points arranged on a Z-order curve. With effective block-based dominance tests, ZSearch can efficiently assert if a block of data points is dominated by a single data point or a block of skyline points. This significantly improves both the overall processing time and the runtime memory consumption.

2.2.2 *Skyline result updates*

Skyline result update is another challenging research problem. In general, dataset updates include insertions and deletions. Insertions of a new data point may bring in new skyline points. In BBS-Update [22],[3] a new data point is compared with the dominance regions of existing skyline points. If the data point is dominated, it is not included in the skyline result. Otherwise, it is admitted to the main memory R-tree. Thereafter, those existing skyline points dominated by this new point are removed. However, main memory R-trees are inefficient to identify those current skyline points dominated by a given new data point. For example, in Fig. 3b, p is a new skyline point and its dominance region intersects with B_a and B_b, implying that some of their enclosed data points may be dominated by p. Thus, p has to be compared with *all* of the enclosed data points. Our ZInsert algorithm utilizes the ordering property of Z-order curves to quickly figure out portions of skyline points possibly dominated by a newly inserted point, and to identify portions potentially dominating the inserted point. Besides, it can support result update with multiple insertions simultaneously.

Deletions are more complicated to handle than insertions since data points previously dominated by a deleted skyline point may no longer be dominated and thus need to be promoted into a skyline result. BBS-Update defines Exclusive Dominance Region (EDR) for each skyline point, i.e., the region inside which all the points are exclusively dominated by the corresponding skyline point. However, in high dimensional spaces, EDRs are in irregular and complex shapes. To efficiently determine whether a data point p (or an index node) is exclusively dominated by the deleted point, DeltaSky [31], an extension of BBS-Update, maintains d sorted lists, each corresponding to one dimension. Each sorted list orders the skyline points according to their values on a respective dimension. If p is dominated, its dominating skyline point(s) should be smaller than or equal to p in all the d sorted lists. Based on this property, DeltaSky performs a negative test. It scans all the lists and determines if all skyline points are greater than p for all the lists. However, DeltaSky also suffers from a serious scalability problem to high data dimensionalities. First, it needs to scan all of the d lists in a high-dimensional space. The number of skyline points is expected to be large [5], which considerably extends the length of all sorted lists. Second, DeltaSky does not address insertions. Sorted lists favor deletions but they incur update overheads to insertions. Our ZDelete algorithm can efficiently find out data points for promotion when some existing skyline points are deleted. More importantly, it can handle batched deletions that none of the existing works can support. Further, our ZUpdate algorithm, which is made up of both ZInsert and ZDelete algorithms, can efficiently handle both insertions and deletions as well as multiple updates simultaneously. The details will be discussed in Sect. 5.

2.2.3 *Skyline query variants*

Based on revised search criteria and dominance conditions, several skyline query variants have been recently proposed to retrieve "good" data points in different perspectives. Here, we discuss four major variants, namely, *skyband queries*, *top-ranked skyline queries*, *k-dominant skyline queries* and *subspace skyline queries*

Skyband query processing. A skyband query retrieves data points dominated by no more than b other data points. In Fig. 4a that illustrates a skyband query example, p_8 is dominated by p_1 and p_3 and hence its count of dominating points is 2, as shown in the associated brace. Based on these counts, a skyband query with b set to 2 returns $\{p_1, p_2, p_3, p_5, p_6, p_7\}$.

[3] We use BBS-Update to refer to the update algorithm derived from the BBS algorithm.

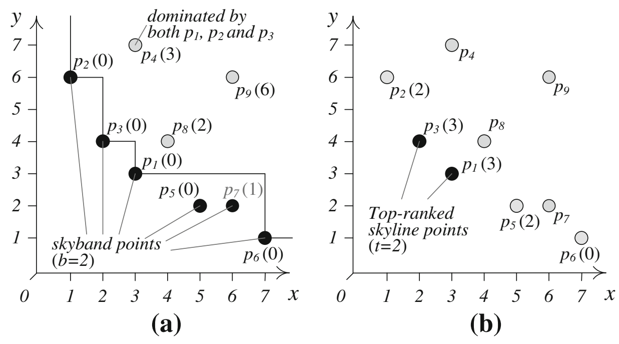


Fig. 4 A skyband query and a top-ranked skyline query. **a** a skyband skyline query, **b** a top-ranked skyline

The other data points, p_4, p_8 and p_9 are excluded as they are dominated by 3, 2, and 6 other data points, respectively.

It is quite straightforward to extend existing skyline algorithms such as SFS [8] and BBS [22] to keep track of data points not dominated by no more than b data points as skyband candidates. However, increasing b to a larger value will definitely result in more skyband candidates and thus incur a higher computational cost in dominance tests. Our ZBand algorithm improves the skyband search efficiency based on the advantage of block-based dominance tests. Specifically, we associate each block of skyband candidates with a *count* to indicate enclosed skyband candidates. If an examinee (that could be a data point or a block of data points) is found to be dominated by some blocks of skyband candidates and the sum of their counts greater than or equal to b, the examinee is certainly dominated by b or more data points and it can be safely discarded. However, the idea of associating counts is not applicable to main-memory R-trees, thus unable to improve BBS performance. The detailed description of the ZBand algorithm is in Sect. 6.2.

Top-ranked skyline query processing. To quantify the importance of a data point p based on the idea of the dominance relationship, as suggested in [22], the number of data points that are dominated by p is used to indicate the importance of p. The more data points p dominates, the more important p is considered to be. Based on this metric, K-dominating query has been proposed [22,33] to retrieve those data points that dominate the most data points. However, K-dominating query returns data points that can be dominated by some other points. Strictly speaking, this K-dominating query is not a skyline query variant. Also, a research work in [20] finds the k most representative skyline points whose dominated data points cover most (if not all) of the dominated data points in a dataset. In [27], a subset of skyline points are collected that can approximately present the distribution of an entire set of skyline points. However, these works cannot tell which skyline points can dominate the most. In this paper, we present a *top-ranked skyline query* that returns a subset of skyline points that can dominate most of other data points. In Fig. 4b, we can see that p_1 dominates p_4, p_8, and p_9 so that its number of dominated data points is 3 (as marked inside the brace). Similarly, the number of data points dominated by p_2, p_3, p_5 and p_6 are 2, 3, 2, and 0, respectively. Based on these numbers, p_1 and p_3 are considered to be the most important while p_6 is the least since it dominates zero data point. Given a parameter t, a top-ranked skyline query finds t skyline points whose counts of dominated data points are the largest. In Fig. 4b, a top-ranked skyline query with $t = 2$ returns p_1 and p_3.

Fig. 5 An example for k-dominant and subspace skyline queries

	s_1	s_2	s_3
p_1	9	11	2
p_2	2	11	11
p_3	8	8	8
p_4	1	25	1

Existing skyline search algorithms can be extended to support top-ranked skyline queries by counting the number of points dominated by each skyline point. Correspondingly, dominance tests that decide whether a data point is dominated should also be extended to count the numbers of data points dominated by individual skyline candidates. However, since data points can be dominated by more than one skyline candidate simultaneously, to record the number of dominated data points for all skyline candidates, exhaustive comparisons between each examinee to all skyline candidates are inevitable. Our ZRank algorithm explores clustering properties of Z-order curve to place skyline candidates as blocks. Very often, a block of skyline candidates dominate some common examinees. By keeping track of the numbers of data points dominated by blocks other than individual skyline points, some comparison overheads can be alleviated. The details of our ZRank algorithm are provided in Sect. 6.3.

***k*-Dominant skyline query processing.** By considering arbitrary k among d dimensions, k-dominant skyline queries [5] reduce the size of skyline result set. However, data points can k-dominate each other simultaneously. Figure 5 lists four three-dimensional data points, namely, p_1, p_2, p_3, and p_4, which are all skyline points based on the conventional dominance condition. If we consider a 2-dominant relationship, p_1 and p_2 2-dominate each other. This cyclic dominance relationship violates the transitivity property, making all the existing skyline search algorithms inapplicable.

As reported in [5], Two-Scan-Algorithm (TSA) is currently the most efficient algorithm to k-dominant skyline queries. TSA scans the dataset twice. The first scan collects candidate skyline points, which might include false hits, and the second scan eliminates those false hits. As an example in Fig. 5, p_1 is first picked and it 2-dominates p_2 and p_3 but not p_4. After the first scan, p_1 and p_4 are retained. In the second scan, p_1 is checked against p_2 and gets dominated. It is removed from the candidate set. A data point p_4, not 2-dominated after the second scan, is the 2-dominant skyline. With the relaxed k-dominant relationship, more data points can be dominated. The search space should therefore be significantly shrunk. However, TSA and other existing approaches, without being conscious of this fact, access all individual data points. In the example as shown in Fig. 5, data points p_1, p_2 and p_3 can be grouped as a block and represented by their lower and upper bounds for each dimension (i.e., $(\langle 2, 9\rangle, \langle 8, 11\rangle, \langle 2, 11\rangle)$) in the search space. It is easy to see that p_4 2-dominates the entire block, which eliminates the pairwise dominance tests between p_4 and all the enclosed data points. Our k-ZSearch algorithm (that will be presented in Sect. 6.4) tackles this k-dominant skyline query problem by exploiting the clustering property of Z-order curve. Whenever a cluster of data points is found to be k-dominated, the examination of all the enclosed points is avoided, thus speeding up the search.

Subspace skyline query processing. To address this subspace skyline search problem, SUBSKY [29] has been recently proposed. It presorts a dataset based on records' minimum attribute values, and it sequentially scans the sorted dataset until the MiniMax of all examined records's attribute values is smaller than the minimum attribute values of all the remaining unexamined records or the dataset is completely scanned. To improve the search efficiency, it groups data points into regions. An entire region is skipped from detailed examinations if it is found to be dominated by the MiniMax of the existing skyline points.

However, SUBSKY can be very inefficient especially when the number of skyline points is large. First, since early examined data points may get dominated by those data points that are accessed later, SUBSKY needs to preserve a buffer of skyline candidates and maintains the buffer when a new skyline candidate is collected. Suppose that a subspace skyline query concerning s_2 is issued on data points illustrated in Fig. 5. First, p_4 (with its minimum attribute value of 1) is first accessed but later it is replaced with p_1 and p_2 whose minimum attribute values are both 2. Then p_1 and p_2 are removed as p_3, the skyline point, is accessed. Thus, this sorting is not appealing. Also, SUBSKY adopts point-to-point dominance tests and the effectiveness of its pruning degrades when many data points cannot be dominated. Instead, our ZSubspace algorithm as will be presented in Sect. 6.5 utilizes the block-based dominance test to enhance the search efficiency. For example, for the same set of data points, p_1, p_2 and p_4 are bounded by a block $(\langle 1, 9\rangle, \langle 11, 25\rangle, \langle 1, 11\rangle)$. Suppose that p_3 is accessed first. By projecting values on s_2, these points are bounded by $(\langle 11, 25\rangle)$. From this, we can quickly assert that the block is dominated by p_3 without exploring it. Further, accessing data points along Z-addresses usually reaches data points close to the origin of the data space prior to others. Those data points typically can dominate many other data points projected on a search subspace.

2.2.4 *Other related works*

Besides, other research studies the evaluations of skyline queries in various environments, e.g., Web [1], distributed databases [32], data stream [19,28], MANET [15], spatial database [25,35], peer-to-peer system [18,30], probabilistic data [23], metric space [7,10], etc. In this paper, our focus is mainly on developing an efficient framework for skyline queries and updates in centralized databases.

3 Z-SKY framework based on Z-order

In this section, we first state the properties of Z-order curves that are found to match perfectly well with the dominance relationship among data points in a data space. This observation inspires our Z-SKY framework development. We then present *ZBtree*, an index structure developed based on the idea of Z-order curves that facilitate skyline searches and result updates as well as index manipulations.

3.1 Skyline query and Z-order curve

The efficiency of skyline query processing and skyline result updating is highly dependent on two factors, i.e., the *access order of the data points* and the *organization of skyline candidates* that facilitates *dominance tests*. An appropriate access order can early identify skyline points that dominate lots of other data points. Thus, this can eliminate unneeded dominance tests and candidate reexaminations [8]. It is desirable to adopt block-based dominance tests if possible, instead of pairwise data point comparisons. To enable block-based dominance tests, data points should be clustered. As will be shown below, Z-order curves can satisfy all those needs.

Figure 6a depicts our nine example 2D data points, i.e., $p_1, \ldots p_9$. Suppose we partition the entire space into four equal-sized quadrants, i.e., regions I, II, III and IV, along the directions parallel to the two axis. Data points in Region I are

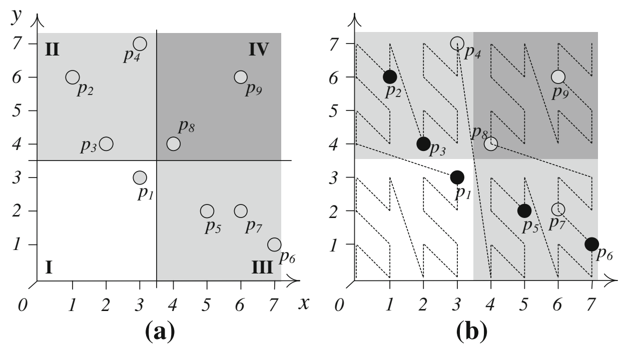


Fig. 6 A Z-order curve example. **a** nine points in a 2D data space, **b** a Z-order curve

definitely not dominated by those in the other three regions. On the contrary, all data points in Region IV are dominated by any data point in Region I. Meanwhile, Region II and III are diagonally opposite to each other, and their data points are incomparable. Dominance tests between them are not needed. However, data points in Regions II or III may get dominated by some data points in Region I while they may dominate some data points in Region IV. Finally, those in Region IV cannot dominate any data point in the other three regions.

These observations not only provide excellent heuristics for dominance tests in regional (i.e., block) level, but also lead to a natural access order of the regions (and their data points) for processing skyline queries, i.e., Region I should be accessed first, followed by Region II, then Region III (or Region III, then Region II) and finally Region IV.[4] The same principles are also applicable to subregions divided from each region. The finest subregion can be just a single coordinate in the space. This access sequence exactly follows a (rotated) 'Z' order, which is the well known Z-order space filling curve. Figure 6b shows the Z-order curve, which starts at the origin and passes through all coordinates (and data points) in the space.

A Z-order curve maps multi-dimensional data points into a one-dimensional space, with each point represented by a unique position, called *Z-address*. A Z-address is a bit string calculated by interleaving the bits of all the coordinate values of a data point. For a d-dimensional space with ([0, 2^v-1]) as the coordinate value ranges, the Z-address of a data point contains dv bits, which can be considered as v d-bit groups. The ith bit of a Z-address is contributed by the (i/d)th bit of the $(i\%d)$th coordinate.[5] In our example, the Z-address of p_2 (1,6) (i.e., ($\mathbf{0}\underline{0}1$,$\mathbf{1}\underline{1}0$) in binary) is $\mathbf{01}\underline{01}10$. Here, **01**, $\underline{01}$, 10 are the three 2-bit groups. Similarly, the Z-address of p_4 (3,7) (i.e.,($\mathbf{0}\underline{1}1$,$\mathbf{1}\underline{1}1$) in binary) is $\mathbf{01}\underline{11}11$ and **01**, $\underline{11}$, 11 are the three 2-bit groups. This Z-address calculation is reversible so that original coordinates in the multi-dimensional space can be recovered by distributing bit values to corresponding dimensions.[6]

Besides, Z-addresses are hierarchical in nature. Given a Z-address with v d-bit groups, the first bit group partitions the search space into 2^d equal-sized sub-spaces, the second bit group partitions each sub-space into 2^d equal-sized smaller sub-spaces, and so on. For instance, p_2 and p_4 share the same first bit group (i.e., 01), and hence both of them fall inside the left part along x-axis and upper part along y-axis (i.e., the Region II in Fig. 6a). With data points arranged according to their Z-addresses, Z-order curves bear two important properties, *monotonic ordering* and *clustering*, as stated below that perfectly match transitivity and incomparability properties of skyline queries, respectively, as already discussed in Sect. 2.1.

Property 3 Monotonic Ordering. *Data points ordered by non-descending Z-addresses are monotonic such that data points are always placed before their dominated points.*

Property 4 Clustering. *Data points with identical prefixes (i.e., leading bit groups) in their Z-addresses are naturally clustered as regions.*

As shown in Fig. 6b, p_1 that dominates p_8 and p_9 is accessed before both of them on the Z-order curve. Similarly, p_2 and p_3 are accessed before p_4, while p_5 is accessed before p_7. This access order guarantees that no candidate reexamination is needed. Due to the hierarchical property of Z-addresses, data points located in the same regions have the same prefixes (i.e., some beginning d-bit groups) in their Z-addresses. As in our example, p_2, p_3 and p_4 (with the first bit-group (i.e., 01) in common) are located inside Region II. Grouping data points in regions can facilitate block-based dominance tests.

Notice that other space filling curves such as Hilbert and Peano curves are inappropriate for skyline query processing as they lack monotonic ordering property. These curves do not always start at the origin of a space (or a subspace), that implies dominating points may be placed after their dominated points, and so candidate reexamination is required.

3.2 ZBtree index structure

The design goals of an index to support skyline query processing and updates are (i) to facilitate data processing along a Z-order address sequence; and (ii) to preserve data points in regions to enable efficient search space pruning. While Z-addresses are one-dimensional, a straightforward approach is to combine Z-order curve and B^+-tree. Thus, we propose *ZBtree*, a variant of B^+-tree, to organize data points in accordance with monotonic Z-addresses. ZBtree indexes disjoint Z-order curve segments (i.e., sequences of data points on the curve), with each corresponding to a region to preserve the clustering property.

In detail, leaf nodes in a *ZBtree* maintain the IDs and the Z-addresses of data points and non-leaf nodes maintain the pointers to their children and the child Z-address intervals (denoted by $[\alpha, \beta]$), each representing a curve segment covering data points with Z-addresses bounded in between α and β. The space spanned by a Z-order curve segment is called a *Z-region*. Figure 7a illustrates a Z-region spanned by a curve starting at point p_8 and ending at point p_9. Since a Z-region can be of any size and in any shape, we bound a Z-region with an RZ-region, as defined in Definition 9.

[4] The visiting order of Region II and III does not affect the correctness of the skyline result.

[5] '/' and '%' are divider and modulus operators, respectively.

[6] We may use Z-address and coordinate interchangeably when the context is clear.

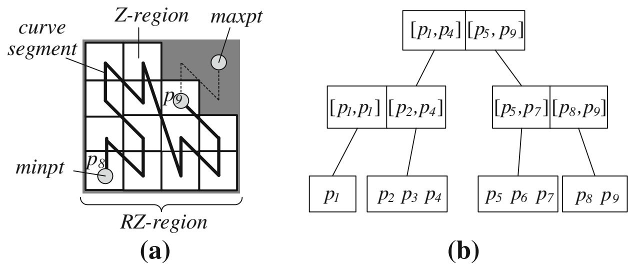


Fig. 7 An RZ-region and a ZBtree. **a** an RZ-region enclosing p_8 and p_9, **b** a ZBtree

Definition 9 RZ-region. An RZ-region is the smallest *square* region covering a Z-region bounded by $[\alpha, \beta]$. The RZ-region is specified by two Z-addresses, $minpt$ and $maxpt$, so $[\alpha, \beta] \subseteq [minpt, maxpt]$. Both $minpt$ and $maxpt$ share an identical prefix d-bit group.

RZ-Regions can be straightforwardly derived based on Z-Regions (i.e., $[\alpha, \beta]$). First, the common prefix of both α and β is determined. With the common prefix, $minpt$ is formed by appending all 0's to the rest of the bits and $maxpt$ is formed by setting all the remaining bits to 1's. Figure 7a shows an RZ-region derived from a Z-region that is covered by a Z-order curve segment in between points p_8 and p_9. While their formations are straightforward, RZ-regions provide the following bounding property that can considerably facilitate dominance tests between RZ-regions.

Property 5 RZ-region bounds. *Within an RZ-region, all data points (except those at* $minpt$*) should be dominated by any data point at* $minpt$*; and data points located at* $maxpt$ *should be dominated by other data points in the region.*

Other works such as [21,24] designed to support multidimensional range searches also exploit the idea of indexing data points on a Z-order curve with a B$^+$-tree. In order to reduce false hits for range searches, their optimization goal is to index Z-order curve segments, each providing the smallest area, which is referred to as Z-region in our discussion. For instance, a node that contain data points p_7, p_8 and p_9 (see Fig. 6a) is very likely to be grouped. Without a need to deal with dominance tests, those existing works may even group incomparable data points in a node. In fact, those data points that forms a small RZ-regions should be grouped in a node so as to facilitate dominance tests. Therefore, the existing approaches cannot be used to support skyline searches efficiently. As we can see, if p_8 and p_9 form a node and this node can be skipped from a detailed examination once p_1 is identified. Thus, different from existing works, our approach is aware of dominance relationships while grouping data points in a *ZBtree*.

3.3 ZBtree index manipulation

Certainly, the search efficiency improvement depends on how data points are organized to form RZ-regions. There are two main index construction objectives. The first objective is to optimize the storage. We can achieve this by packing as many data points (or nodes) as possible in leaf nodes (or non-leaf nodes, respectively) in a *ZBtree* such that the size of the *ZBtree* is minimized. In case that an entire index is needed to traverse, the access overhead can be reduced. The second objective is to optimize the retrieval performance. We can strategically allocate data points that provide small RZ-regions stored in nodes, such that search space pruning and dominance tests can be effectively facilitated. As shown in Fig. 6b, a node that contains data points p_8 and p_9 can be discarded once we identify the data point p_1, and a node that contains data points p_2, p_3, and p_4 does not need to be examined against another node that contains data points p_5, p_6, and p_7. Consider that the node capacities range from 1 to 3. p_1 can be put into the first leaf node. Next, p_2, p_3 and p_4 are inserted into the second leaf node. Similarly, p_5, p_6 and p_7 occupy the third leaf node. Finally, p_8 and p_9 are put into the last leaf node. This index structure is depicted in Fig. 7b. While this requires some extra storage, this region-aware grouping improves the search performance, as some unnecessary node traversal and comparisons between incomparable nodes are avoided. Based on the same principle, we group leaf nodes into appropriate non-leaf nodes and recursively propagate the process upwards till the root of the index is formed.

In the following, we assume that each node can contain at most N entries and the minimum node utilization threshold is M (where $M \leq N/2$) and describe ZBtree manipulations such as insertion, deletion and bulkloading.

ZBtree insertion. Insertion places a data point with a Z-address z_{ins} to a leaf node in a *ZBtree*. The search for a target leaf node to accommodate the new data point is based on a depth-first search with z_{ins} as the search key. Along the traversal, the branch whose $[\alpha, \beta]$ covers z_{ins} is explored. In case that no branch with an interval covering the data point is identified, a branch with the smallest resulted RZ-region is chosen.[7] After an insertion, a leaf node filled with more than N entries becomes overflow and needs to be split into two new nodes. These two nodes are formed to cover two disjointed Z-order curve segments such that the total sizes of their corresponding RZ-regions are the smallest among all possible splits. After an insertion, the interval of the inserted leaf node is updated to accommodate the newly inserted data point. This interval update is propagated up from the leaf node to its parent and ascendent nodes.

[7] A resulted RZ-region here is referred to as the extended RZ-region after inserting the new data point.

ZBtree deletion. Deletion removes a data point with Z-address z_{del} from a *ZBtree*. First, a leaf node that contains the deleted data point is found by a depth-first search using z_{del} as the search key. The data point is then removed from the node. In case that the leaf node, after deletion, contains less than M entries, it needs to be removed. All the enclosed data points are re-inserted into a *ZBtree* based on the above-described insertion procedure. Similarly, the intervals of the nodes along the path from affected leaf nodes to the root node are updated to reflect the deletion.

ZBtree bulkloading. Bulkloading builds a *ZBtree* in a bottom-up fashion. It is mainly used to prepare a source dataset for skyline queries. It has two steps. The first step sorts all data points based on Z-addresses. The second step scans these data points (or nodes) following the ascending Z-addresses with a sliding window of N slots to form leaf nodes (or non-leaf nodes, respectively). Since data points in a high dimensional space are very often sparsely distributed, it is very unlikely that small RZ-regions can be formed by including a lot of points in a node. Then, such small RZ-regions can be formed but sacrificing the space utilization. Also, as the number of skyline points is expected to be large, many RZ-regions may not be dominated and thus, the entire dataset needs to be examined. While various bulkloading strategies are examined and no significant difference is observed in terms of the overall processing time [17], compact bulkloading strategy is adopted in this work to maximize the node utilization and minimize the access cost. It sorts all data points in an ascending order of their Z-addresses and forms leaf nodes based on every N data points. Similarly, it puts every N leaf nodes together to form non-leaf nodes until the root of a *ZBtree* is formed.

4 Skyline query processing

In this section, we discuss *ZSearch*, an efficient skyline search algorithm. Its efficiency is attributed to RZ-region based dominance tests and search space pruning.

4.1 RZ-region based dominance test

Dominance tests are a key determinant of computational overhead in skyline query processing. To avoid comparing data points in a time-consuming pairwise fashion, we introduce block based dominance tests, which are based on RZ-regions derived during traversing a *ZBtree*. In the Z-SKY framework, we maintain a source dataset $\mathcal{SRC}$ and a set of skyline candidates $\mathcal{SL}$ both indexed by *ZBtree*s. These two indexes present clusters of data points in RZ-regions. The dominance relationships between any two RZ-regions are defined in Lemma 1. When not causing any confusion, we

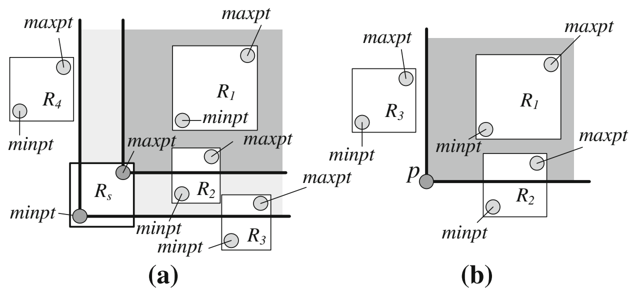


Fig. 8 Examples of dominance tests based on RZ-regions. **a** two RZ-regions, **b** RZ-regions versus a data point

use RZ-region R to denote all the data points within it and use $minpt(R)$ and $maxpt(R)$ to represent the $minpt$ and $maxpt$ of R, respectively.

Lemma 1 *Given two RZ-regions R and R', the following four cases hold:*

1. *If $maxpt(R) \vdash minpt(R')$, then $R \vdash R'$, then any data point in R for sure dominates all the data points in R'.*
2. *If $maxpt(R) \nvdash minpt(R') \wedge minpt(R) \vdash minpt(R')$, then some data points in R may dominate all the data points in R'.*
3. *If $minpt(R) \nvdash minpt(R') \wedge minpt(R) \vdash maxpt(R')$, then it is possible that some data points in R dominates some data points in R'.*
4. *If $minpt(R) \nvdash maxpt(R')$, then $R \nvdash R'$, then no data point in R can dominate any data point in R'.* □

Proof We prove the lemma case by case.
Case 1. $maxpt(R) \vdash minpt(R')$. By the transitivity property (Property 1) and RZ-region bounds (Property 5), $\forall p \in R - \{maxpt(R)\}, \forall p' \in R' - \{minpt(R')\}, p \vdash maxpt(R) \wedge minpt(R') \vdash p' \Rightarrow p \vdash p'$ and as stated, $maxpt(R) \vdash minpt(R')$. Hence, $R \vdash R'$. R_s and R_1 in Fig. 8a form an example.
Case 2. $maxpt(R) \nvdash minpt(R') \wedge minpt(R) \vdash minpt(R')$. Data points (e.g. $minpt(R')$) are dominated by $minpt(R)$. Thus, the case holds. R_s and R_2 in Fig. 8a form an example.
Case 3. $minpt(R) \nvdash minpt(R') \wedge minpt(R) \vdash maxpt(R')$. As some data points in R (e.g. $maxpt(R')$) are dominated by $minpt(R)$, the case holds. R_s and R_3 in Fig. 8a form an example.
Case 4. $minpt(R) \nvdash maxpt(R')$. This case can be proved by contradiction. Assume $p \vdash p'$ (where $p \in R$, $p' \in R'$). By transitivity, $minpt(R) \vdash p \vdash p' \vdash maxpt(R') \Rightarrow minpt(R) \vdash maxpt(R')$. This contradicts the condition of the case. R_s and R_4 in Fig. 8a form an example. □

This dominance relationship is generic enough to be applied to an RZ-region, which contains a single data point

```
Function $Dominate(\mathcal{SL}, minpt, maxpt)$
Input.   $\mathcal{SL}$: a ZBtree indexing skyline points;
         $minpt$ and $maxpt$: endpoints of an RZ-region;
Local.   $q$: Queue;
Output.  TRUE if input is dominated, else FALSE;
Begin
 1.  $q$.enqueue($\mathcal{SL}$'s root);
 2.  while ($q$ is not empty) do
 3.    var $n$: Node;
 4.    $q$.dequeue($n$);
 5.    if ($n$ is a non-leaf node) then
 6.      forall (child node $c$ of $n$) do
 7.        if ($c$'s RZ-region's $maxpt \vdash minpt$) then
 8.          output TRUE;        /* Case 1 of Lemma 1 */
 9.        else if ($c$'s RZ-region's $minpt \vdash minpt$) then
10.          $q$.enqueue($c$);      /* Case 2 of Lemma 1 */
11.    else                          /* leaf node */
12.      forall (child point $c$ of $n$) do
13.        if ($c \vdash minpt$) then
14.          output TRUE;
15.  output FALSE;
End.
```

Fig. 9 The pseudo-code of the *Dominate* function

```
Algorithm $ZSearch(\mathcal{SRC})$
Input.   $\mathcal{SRC}$: a ZBtree for source data set;
Local.   $s$: Stack;
Output.  $\mathcal{SL}$: a ZBtree for skyline candidates;
Begin
 1.  $s$.push($\mathcal{SRC}$'s root);
 2.  while ($s$ is not empty) do
 3.    var $n$: Node;
 4.    $s$.pop($n$);
 5.    if ($Dominate(\mathcal{SL}$, $n$'s $minpt$, $n$'s $maxpt$)) then
 6.      goto line 3.
 7.    if ($n$ is a non-leaf node) then
 8.      forall (child node $c$ of $n$) do
 9.        $s$.push($c$);
10.    else /* leaf node */
11.      forall (child point $c$ of $n$) do
12.        if (not $Dominate(\mathcal{SL}$, $c$, $c$)) then
13.          $\mathcal{SL}$.insert($c$);
14.  output $\mathcal{SL}$;
End.
```

Fig. 10 The pseudo-code of the *ZSearch* algorithm

p, i.e., $minpt(R) = maxpt(R) = p$, and/or $minpt(R') = maxpt(R') = p'$. Fig. 8b shows the comparisons of R_1, R_2, R_3 against p. Further, the dominance relationship between two data points (as stated in Definition 1 in Sect. 2.1) becomes a special case of this lemma.

Based on Lemma 1, we can perform effective dominance tests on the RZ-region of any node from $\mathcal{SRC}$ against those from $\mathcal{SL}$. Figure 9 outlines *Dominate* function for the dominance test. It checks whether a given RZ-region R from $\mathcal{SRC}$ (represented by its $minpt$ and $maxpt$)[8] contains any potential skyline points by conducting dominance tests against all the identified skyline points in $\mathcal{SL}$. The function traverses $\mathcal{SL}$ based on *breadth-first traversal* such that the RZ-regions of upper-level nodes from $\mathcal{SL}$ are compared against R first and drilled down if R needs further examination against finer RZ-regions in $\mathcal{SL}$.

In the function, a queue is initialized with the root of $\mathcal{SL}$. An entry n popped from the queue will be further explored in two situations:

- n is a non-leaf node, and all its child entries c are examined. If any point inside the RZ-region of c dominates R (i.e., Case 1 of Lemma 1), the algorithm indicates that R is dominated and the dominance test is completed (lines 7–8). If some points inside the RZ-region of c may dominate R (i.e., Case 2 of Lemma 1), the entry c is enqueued for further examination (lines 9–10). Tests on Case 3 and Case 4 of Lemma 1 are omitted since they are implied by the failures of the above two cases and the input RZ-region should not be completely dominated or these two RZ-regions are found to be incomparable.
- n is a leaf node. We check R against individual skyline points inside n (lines 13–14).

[8] In case of a data point, p, R contains p and $minpt(R) = maxpt(R) = p$.

This traversal continues until the queue is vacated (i.e., all comparable nodes in $\mathcal{SL}$ are visited). Finally, R is reported to be not dominated. Notice that the function may stop before the leaf level is reached. As such, a few nodes in $\mathcal{SL}$ need to be accessed and thus an efficient block-based dominance test is provided.

4.2 The *ZSearch* algorithm

The *ZSearch* algorithm traverses $\mathcal{SRC}$ to examine RZ-regions and data points in a *depth-first* order, which exactly follows the order of Z-addresses, with a stack keeping track of unexplored paths. The stack memory consumption is bounded by a factor of the tree height of $\mathcal{SRC}$. Figure 10 depicts the pseudo-code of the *ZSearch* algorithm. It fetches the index nodes and/or data points from $\mathcal{SRC}$ (lines 2–13). At each round, the RZ-region of a node is examined against $\mathcal{SL}$ with the *Dominate* function (see Fig. 9). If its corresponding RZ-region is not dominated, the node is further explored (lines 7–13). Data points not dominated by any skyline candidate in $\mathcal{SL}$ are collected as new skyline candidates and appended to $\mathcal{SL}$ (line 13). The mechanism of appending skyline candidates to $\mathcal{SL}$ is based on the ZBtree insertion (discussed in Sect. 3.3). As the newly admitted skyline candidates should be added to the latest created leaf node since their Z-addresses are greater than those of collected skyline candidates, we maintain a pointer to the latest created leaf node to save the traversals in $\mathcal{SL}$.

Stack	Skyline points S L
[p1 , p4],[p5 , p9]	∅
[p1 , p1],[p2 , p4],[p5 , p9]	∅
[p2 , p4],[p5 , p9]	{p1 }
[p5 , p9]	{p1 },{p2 , p3 }
[p5 , p7],[p8 , p9]	{p1 },{p2 , p3 }
[p8 , p9]	{p1 },{p2 , p3 },{p5 , p6 }

Fig. 11 The trace of the $ZSearch$ algorithm

The algorithm terminates when all entries from $\mathcal{SRC}$ are examined (i.e., signaled by an empty stack). The organization of skyline candidates in $\mathcal{SL}$ enables the formation of RZ-regions to facilitate dominance tests. $\mathcal{SL}$ may be too large to fit in the main memory. It can be stored on disk and we use available main memory as a cache. In this case, our approach is very appropriate since (1) the cached upper (and usually a small) portion of $\mathcal{SL}$ can be sufficient to perform dominance tests as the leaf levels may not always be reached and (2) clustered data points in $\mathcal{SRC}$ exhibit good access locality of related branches in $\mathcal{SL}$. Finally we use Example 1 to illustrate how the *ZSearch* algorithm runs.

Example 1 We use the example data points in Fig. 6a indexed by a corresponding *ZBtree* in Fig. 7b to illustrate the *ZSearch* algorithm. The trace is depicted in Fig. 11. Initially, the root entries, i.e., $[p_1, p_4]$ and $[p_5, p_9]$, are pushed to the stack and $\mathcal{SL}$ is empty. For presentational simplicity, only leaf nodes enclosed by {} are shown. First, we obtain p_1, the first skyline point. Next, p_2, the second accessed data point, not dominated by p_1, is inserted to $\mathcal{SL}$. Notice that BBS and SFS access p_3 after p_1 but before p_2. Then, the delivery order of a skyline result by the *ZSearch* algorithm is not necessarily the same as BBS and SFS.

Assume that the node capacity of $\mathcal{SL}$ is 2. Insertion of p_3 triggers a node split. p_2 and p_3 are put together since $\{p_1\}$ and $\{p_2, p_3\}$ form smaller RZ-regions. p_4, dominated by p_2 (or p_3), is discarded. Later, $[p_5, p_9]$ is explored. As the RZ-region of $[p_5, p_7]$ is incomparable to $\{p_2, p_3\}$, the comparison with p_2 or p_3 is saved. Next, explored p_5 and p_6 are inserted to $\mathcal{SL}$ but p_7 is dropped. Finally, $[p_8, p_9]$ dominated by $\{p_1\}$ in $\mathcal{SL}$ is skipped and the search completes. □

5 Skyline result maintenance

As opposed to reevaluating a skyline query whenever the underlying dataset is changed, which is clearly inefficient, we present efficient skyline result update algorithms in this section, namely, *ZInsert*, *ZDelete* and *ZUpdate*, that can incrementally update the skyline RESULTS. Our following discussion assumes the updates of $\mathcal{SRC}$ are performed based on the previously discussed *ZBtree* index manipulations right before the update of a skyline result.

```
Algorithm ZInsert($\mathcal{SL}$, $P_{ins}$)
Input.  $\mathcal{SL}$: ZBtree for skyline points;
        $P_{ins}$: a (ordered) set of new data points
        (based on Z-addresses);
Local.  $s$: Stack;
Begin
 1.  $s$.push($\mathcal{SL}$'s root);
 2.  while ($s$ is not empty and $P_{ins}$ is not empty) do
 3.    var $n$: Node;
 4.    $s$.pop($n$);
 5.    forall ($p \in P_{ins} \mid n$'s $maxpt < p$) do /* dominance test */
 6.      if ($n$'s $maxpt \vdash p$) then
 7.        $P_{ins} \leftarrow P_{ins} - \{p\}$;
 8.    if ($\exists\, p \in P_{ins}, p \vdash n$'s $minpt$) then      /* reexamination. */
 9.      $\mathcal{SL}$.remove($n$); goto line 3;
10.    if ($n$ is a node) then  /* explore $n$'s child nodes or points */
11.      if ($\nexists\, p \in P_{ins}, p \vdash n$'s $maxpt$ and $n$'s $minpt \vdash p$) then
12.        goto line 3;                              /* no need to expand $n$ */
13.      else
14.        forall (child $c$ of $n$) do
15.          $s$.push($c$);
16.  forall ($p \in P_{ins}$) do
17.    $\mathcal{SL}$.insert($p$);
End.
```

Fig. 12 The pseudo-code of the *ZInsert* algorithm

5.1 The *ZInsert* algorithm

Inserting a new data point into $\mathcal{SRC}$, p_{ins}, which may be a new member of a skyline result, involves two checks on: (i) whether p_{ins} is a skyline point that involves dominance tests against some of existing skyline points and (ii) whether p_{ins} dominates any existing skyline point if p_{ins} passes the dominance test (i.e., the first check) that triggers candidate reexaminations. Figure 12 lists the pseudo-code of the ZInsert algorithm. In general, its idea is similar to *Dominate* function in Fig. 9 but it considers a set of new inserted data points, P_{ins} in order to share the skyline update overheads among multiple insertions simultaneously. The ZInsert algorithm has a main loop (line 2-15) to access $\mathcal{SL}$ with a stack s keeping track of pending unexamined nodes and data points in a non-decreasing Z-address order. The iteration ends when s becomes empty or P_{ins} is vacated. The empty P_{ins} is resulted when all the data points in it are dominated.

In the algorithm, we assume that data points in P_{ins} are incomparable to each other. Whenever an entry (that can be a node or a data point) is popped from a stack s and investigated, the first check (i.e., a dominance test) is performed against individual data points in P_{ins}. A data point p is removed from P_{ins} if it is dominated by a data point or the $maxpt$ of an RZ-region of a node in $\mathcal{SL}$ (line 5–7). Thereafter, the second check (i.e., candidate reexamination) takes place such that those nodes or data points are removed from $\mathcal{SL}$ if they are dominated by any data point in P_{ins} (lines 8–9). Besides, we skip exploring this node once it is discarded. Next, a node in $\mathcal{SL}$ is further expanded if it may contain data points that can dominate new data points in P_{ins}, or its contained data points are possibly dominated by the

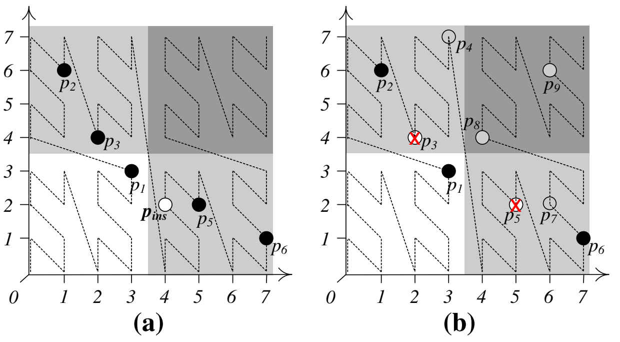


Fig. 13 Updates of a skyline result. **a** insertion of p_{ins}, **b** deletion of p_3 and p_5

new data points in P_{ins}. Otherwise, the node can be safely skipped from a detailed examination (lines 10–15). At last, all remaining data points in P_{ins} that are not dominated by existing skyline candidates are inserted to $\mathcal{SL}$ (lines 16–17).

It is noteworthy that, due to the monotonic ordering property of Z-order curve (Property 3), p is guaranteed not to be dominated by those skyline points whose Z-addresses are larger than that of p. Thus in a dominance test, a node n popped from s only needs to be compared with those data points in P_{ins} with their Z-addresses larger than n's. Because of the same reason, candidate reexaminations examine only those skyline points in $\mathcal{SL}$ with Z-addresses smaller than the Z-addresses of inserted data points. Figure 13a explains this useful property in dominance tests and candidate reexaminations. A new data point p_{ins} that locates between p_3 and p_5 on the Z-order curve is inserted. In the dominance test, p_{ins} is only needed to be compared against both p_1 and an RZ-region formed by p_2 and p_3. It passes the dominance test. During the candidate reexamination, only p_5 and p_6 (whose Z-addresses are greater than p_{ins}'s) are reexamined to check if they are dominated by p_{ins}. In this case, p_5 is removed while p_6 is retained.

5.2 The *ZDelete* algorithm

Handling deletion of a data point from a skyline query result is more complicated than insertion. Deleting any existing skyline point, say p_{del}, from a skyline result may get some data points previously dominated by p_{del} promoted to the skyline result. Since only those data points *exclusively* dominated by p_{del} are promoted, the most critical issue is how to efficiently locate them from $\mathcal{SRC}$. To address this, we devise the *ZDelete* algorithm as outlined in Fig. 14. To alleviate the average skyline update cost that is incurred by scanning $\mathcal{SRC}$ multiple times, deletions can be accumulated as P_{del} and performed as a batch. For instance, even though p_3 and p_5 in Fig. 13b have been deleted at different times, the search for promoted skyline candidates due to their deletions can be processed together at a later time.

```
Algorithm $ZDelete(\mathcal{SRC}, \mathcal{SL}, P_{del})$
Input.  $\mathcal{SRC}$: ZBtree for source dataset;
        $\mathcal{SL}$: ZBtree for skyline points;
        $P_{del}$: a (ordered) set of deleted skyline points
          (based on Z-addresses);
Local.  $s$: Stack;
Begin
 1.  $s$.push($\mathcal{SRC}$'s root);
 2.  while $s$ is not empty do
 3.    var $n$: Node;
 4.    $s$.pop($n$);
 5.    if ($\nexists\, p \in P_{del}, p \vdash n$'s $minpt$) or
 6.      ($Dominate(\mathcal{SL}, n$'s $minpt, n$'s $maxpt$)) then
 7.      goto line 3.
 8.    if $n$ is a node then
 9.      forall child node $c$ of $n$ do
10.        $s$.push($c$);
11.    else                          /* $n$ is a data point */
12.        $\mathcal{SL}$.insert($c$);
End.
```

Fig. 14 The pseudo-code of the *ZDelete* algorithm

The algorithm scans $\mathcal{SRC}$ and uses a stack s to keep track of unexamined entries (i.e., nodes or data points). This depth-first traversal aligns the lookup of data points in non-decreasing Z-addresses. The iteration ends when s becomes empty (lines 2–12). The algorithm traverses $\mathcal{SRC}$ and checks nodes and data points against a set of data points P_{del} and $\mathcal{SL}$. Here, $\mathcal{SL}$ is assumed to have all data points in P_{del} removed. The use of the RZ-regions of $\mathcal{SL}$ can alleviate computational overheads incurred by dominance tests.

Whenever an entry n is popped, it is checked if it is exclusively dominated by the deleted data points via two checks on: (i) whether n is dominated by data points in P_{del}; and (ii) whether n is not dominated by any skyline candidate in $\mathcal{SL}$ (lines 5–7). Only those entries who pass these two checks may contain or may be promoted skyline points and need further examination (lines 9–10). If n is a node, all its child entries are pushed into the stack for later exploration; otherwise n must be a data point and it is inserted to $\mathcal{SL}$. Further, owing to the monotonic ordering property of Z-order curves, those dominated by p_{del} should have Z-addresses greater than that of p_{del}. In the algorithm, the initial portion of $\mathcal{SRC}$ (i.e., data points have their Z-addresses smaller than the smallest Z-addresses) in P_{del} is skipped.

5.3 The *ZUpdate* algorithm

In practice, updates involve a mixture of insertions and deletions. It is trivial to handle individual inserted and/or deleted data points according to their update order. However, as for a usual case that deletions and insertions interleave during data points update, the batched process of insertions and deletions would incur multiple traversals of $\mathcal{SL}$ and $\mathcal{SRC}$. To reduce the traversal cost, we propose the *ZUpdate* algorithm (as outlined in Fig. 15) that groups update operations in accordance

```
Algorithm ZUpdate($\mathcal{SRC}$, $\mathcal{SL}$, $P_{upd}$)
Input.    $\mathcal{SRC}$: ZBtree for source dataset;
          $\mathcal{SL}$: ZBtree for skyline points;
          $P_{upd}$: a set of inserted or deleted data points;
Local.    $P_{ins}$: a set of inserted data points;
          $P_{del}$: a set of deleted skyline points;
Begin
1.    $P_{ins} \leftarrow \{p \in P_{upd} | p$ is an inserted data point$\}$;
2.    $ZInsert(\mathcal{SL}, P_{ins})$;    /* Algorithm $ZInsert$ */
3.    $P_{del} \leftarrow \{p \in P_{upd} | p$ is a deleted data point and $p \in \mathcal{SL}\}$;
4.    $ZDelete(\mathcal{SRC}, \mathcal{SL}, P_{del})$;    /* Algorithm $ZDelete$ */
End.
```

Fig. 15 The pseudo-code of the *ZUpdate* algorithm

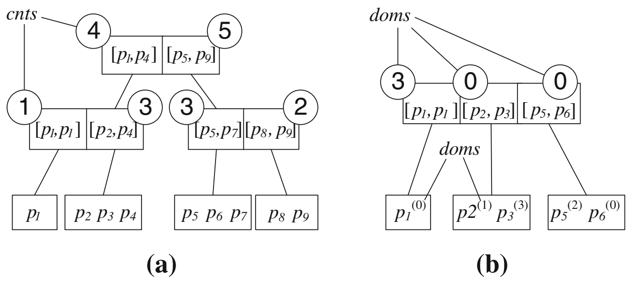


Fig. 16 Example extended ZBtrees. **a** incorporation with *cnts*, **b** incorporated with *doms*

with the types of operations (i.e., insertions and deletions) and performs all the insertions followed by all the deletions.

Since the execution order of deletions and insertions does not affect the correctness of the update if they are on different data points, the *ZUpdate* algorithm performs insertion (i.e., by the *ZInsert* algorithm) prior to deletions (i.e., by the *ZDelete* algorithm) to boost update efficiency. It is because the newly promoted skyline points may be dominated by inserted data points (if the *ZInsert* algorithm is performed after the *ZDelete* algorithm) and handling deleted skyline points is far more expensive than insertions. Due to the transitivity property, whenever deleted skyline points are found to be dominated by any newly inserted skyline point, all other data points dominated by those deleted skyline points must be dominated by the inserted skyline point. Thus, we can simply discard those deleted data points that have already been dominated by any newly inserted data point from the processing. This can effectively eliminate the expensive and redundant search of data points for promotion. Hence update performance can be considerably improved. For instance, in the combined case of those in Fig. 13a and b, handling deletion of p_5 can be eliminated due to the inserted data point p_{ins}. In our algorithm, inserted data points from P_{upd} are first extracted as P_{ins} (line 1) and processed with the *ZInsert* algorithm (line 2). Then the deleted data points from P_{upd} that still appear in $\mathcal{SL}$ are put into P_{del} (line 3) and examined by the *ZDelete* algorithm (line 4). After the completion of the *ZUpdate* algorithm, $\mathcal{SL}$ is updated.

6 Skyline query variants

In this section, we extend our Z-SKY framework to handle different skyline query variants. Although those skyline query variants are defined differently, strategies that can efficiently answer them can be very similar. For skyband and top-ranked skyline queries, counting of dominating and dominated data points is needed in addition to dominance tests. Thus, we extend the *Dominate* function and put some additional information in *ZBtree*s to facilitate the counting. On the other hand, for k-dominant skyline and subspace skyline queries, it is not guaranteed that data points are not dominated by those accessed after them following the access order on their Z-addresses. Thus, we adopt a filter-and-reexamine approach in which candidates that may contain false hits are first retrieved, and then they are reexamined to discard those false hits. Based on the idea of Z-order curves and above techniques, we derive and discuss the algorithms, namely, *ZBand*, *ZRank*, k-*ZSearch* and *ZSubspace* to evaluate skyband queries, top-ranked skyline queries, k-dominant skyline queries and subspace skyline queries, respectively, in the following.

6.1 *Extended ZBtrees*

For some skyline query variants like skyband and top-ranked skyline queries as will be discussed in this section, auxiliary information like the number of dominating points or the number of dominated points provided in nodes in a *ZBtree* can speed up the search. With a plain *ZBtree* that does not have these types of auxiliary information, the search has to traverse the index down to some leaf nodes to determine the exact number of data points. Otherwise, if some auxiliary information like *count* of data points indexed is provided in some high-level nodes, the counting of individual data points can be done in some nodes before accessing the leaf level of an index. In the following, we call *ZBtree*s augmented with auxiliary information as *extended ZBtree*s.

Subject to query needs, we can associate different types of auxiliary information to the nodes of *extended ZBtree*s. The first type of auxiliary information we consider is the number of indexed data points of each node, labeled as *cnt*. Fig. 16a depicts an example of *extended ZBtree* that associates a *cnt* with each node for the dataset as depicted in Fig. 6. These *cnt*s are useful to both skyband queries and top-ranked skyline queries. For example, let n be the node entry $[p_8, p_9]$. A *cnt* $(= 2)$ is associated with n that indicates two data points indexed by the pointed node. As a result, if n is dominated by a data point p, it is certain, without a detailed examination, that p dominates the two data points indexed by n.

The second type of auxiliary information is the number of data points that are dominated by each associated

```
Function $BandDominate(cnt\text{-}\mathcal{SL}, minpt, maxpt, b)$
Input.   $cnt\text{-}\mathcal{SL}$: a extended ZBtree indexing skyband candidates;
         $minpt$ and $maxpt$: endpoints of an RZ-region;
         $b$: threshold skyband width;
Local.   $q$: Queue; $cnt$: a counter of dominating points
Output.  TRUE if input is dominated, else FALSE;
Begin
 1.  $q$.enqueue($cnt\text{-}\mathcal{SL}$'s root); $cnt \leftarrow 0$;
 2.  while ($q$ is not empty) do
 3.    var $n$: Node;
 4.    $q$.dequeue($n$);
 5.    if ($n$ is a non-leaf node) then
 6.      forall (child node $c$ of $n$) do
 7.        if ($c$'s RZ-region's $maxpt \vdash minpt$) then
 8.          $cnt \leftarrow cnt + c$'s $cnt$; /* Case 1 of Lemma 1 */
 9.        else if ($c$'s RZ-region's $minpt \vdash minpt$) then
10.          $q$.enqueue($c$);          /* Case 2 of Lemma 1 */
11.    else /* leaf node */
12.      forall (child point $p$ of $n$) do
13.        if ($p \vdash minpt$) then
14.          $cnt \leftarrow cnt + 1$;
15.    if ($cnt \geq b$) then output TRUE;
16.  output FALSE;
End.
```

Fig. 17 The pseudo-code of the *BandDominate* function

```
Algorithm $ZBand(\mathcal{SRC}, b)$
Input.   $\mathcal{SRC}$: a ZBtree for source data set;
Local.   $s$: Stack;
Output.  $cnt\text{-}\mathcal{SL}$: a extended ZBtree for skyline points;
Begin
 1.  $s$.push($\mathcal{SRC}$'s root);
 2.  while ($s$ is not empty) do
 3.    var $n$: Node;
 4.    $s$.pop($n$);
 5.    if ($BandDominate(cnt\text{-}\mathcal{SL}, n$'s $minpt, n$'s $maxpt, b)$)
 6.      then goto line 3.
 7.    if ($n$ is a non-leaf node) then
 8.      forall (child node $c$ of $n$) do
 9.        $s$.push($c$);
10.    else /* leaf node */
11.      forall (child point $c$ of $n$) do
12.        if (not $BandDominate(cnt\text{-}\mathcal{SL}, c, c, b)$) then
13.          $cnt\text{-}\mathcal{SL}$.insert($c$);
14.  output $cnt\text{-}\mathcal{SL}$;
End.
```

Fig. 18 The pseudo-code of the *ZBand* algorithm

node, labeled as *dom*, useful for top-ranked skyline queries. Because of the clustering property, data points dominated by a node n in $\mathcal{SL}$ are for sure dominated by n's descendants. Therefore, the *dom* associated with n's child entry does not count for those points already dominated by n, and the dominating power, i.e., the number of dominated data points, of a candidate skyline point can be derived by summing up the *dom*s of its enclosing node, its parent node, and all its ancestors. An example extended ZBtree associated with *dom*s is depicted in Fig. 16b. The *dom* associated with the node entry $[p_1, p_1]$ is three. The dominance power of the candidate point p_1 is three that is the summation of the *dom* associated with p_1 (i.e., 0) and the *dom* associated with the node $[p_1, p_1]$ (i.e., 3).

6.2 The *ZBand* algorithm

A skyband query retrieves those data points that are not dominated by more than b other data points in a dataset. Due to the transitivity property, any given data point should appear after all its dominating data points on a Z-order curve. We can naturally extend dominance tests for skyband queries to check whether an examinee is dominated by more than b collected skyband candidates.

However, the number of skyband candidates increases when b grows. Therefore, dominance tests between each examinee data point and all the candidate points may incur a longer precessing time. To improve the search performance, we derive *BandDominate* function as outlined in Fig. 17. While its logic is very similar to *Dominate* function (in Fig. 17), *BandDominate* function uses cnt-$\mathcal{SL}$ to facilitate the counting of dominating points and reports dominated if the number of dominating points is found to be greater than or equal to b.

As outlined in Fig. 17, *BandDominate* function examines an RZ-region R from $\mathcal{SRC}$ against skyband candidates indexed by cnt-$\mathcal{SL}$. A queue initialized with the root node of cnt-$\mathcal{SL}$ is used to guide the breadth-first traversal on cnt-$\mathcal{SL}$. In addition, a count cnt with initial value set to 0 is used to record the number of data points found so far that dominate R (line 1). Thereafter, the search proceeds by iteratively fetching the head entry n from the queue (lines 2–15). If n is a non-leaf node, we check for its every child entry c. If RZ-region of c dominates $minpt$ (and hence the entire R), we increment cnt by c's cnt (lines 7–8). Or, if c may contain some data points dominating R, it is enqueued for further exploration (line 9-10). On the other hand, n must be a leaf node and all the enclosed data points are compared against R. Each detected dominating data point increases cnt by one (lines 12–14). By the end of each iteration, we check the early termination condition, i.e., whether cnt reaches b, that indicates R is for sure dominated by b or more than b data points (line 15) to terminate the algorithm. With cnt's presented in the immediate nodes, cnt-$\mathcal{SL}$ is not needed to be traversed to the bottom and hence the performance is improved.

Figure 18 sketches the *ZBand* algorithm. Like the *ZSearch* algorithm, it traverses $\mathcal{SRC}$ in depth-first order, but it uses cnt-$\mathcal{SL}$ to maintain skyband candidates and the *BandDominate* function to perform dominance tests, in place of $\mathcal{SL}$ and *Dominate* function, respectively. The logic is very similar to the *ZSearch* algorithm. Finally, we use Example 2 to illustrate the operation of the *ZBand* algorithm.

Example 2 This example illustrates how the *ZBand* algorithm evaluates a skyband query with $b = 2$ on our example dataset. Figure 19 outlines the trace. We present a node in cnt-$\mathcal{SL}$ as $\{p\}^{(cnt)}$ that means the node has a data point p and it is associated with a count cnt of enclosed data points.

Stack	Skyline points cnt-$\mathcal{SL}$
$[p_1, p_4]$,$[p_5, p_9]$	$\emptyset$
$[p_1, p_1]$,$[p_2, p_4]$,$[p_5, p_9]$	$\emptyset$
$[p_2, p_4]$,$[p_5, p_9]$	$\{p_1\}^{(1)}$
$[p_5, p_9]$	$\{p_1\}^{(1)}$,$\{p_2, p_3\}^{(2)}$
$[p_5, p_7]$,$[p_8, p_9]$	$\{p_1\}^{(1)}$,$\{p_2, p_3\}^{(2)}$
$[p_8, p_9]$	$\{p_1\}^{(1)}$,$\{p_2, p_3\}^{(2)}$, $\{p_5, p_6, p_7\}^{(3)}$

Fig. 19 The trace of the *ZBand* algorithm

Initially, the stack is initialized with the root node of $\mathcal{SRC}$, i.e., branches $[p_1, p_4]$ and $[p_5, p_9]$. Then, the first leaf node $[p_1, p_1]$ is explored and the first data point, p_1, is pushed into cnt-$\mathcal{SL}$. Next, the second leaf node $[p_2, p_4]$ is explored. The first two data points p_2 and p_3 are examined; they are not dominated and enrolled into cnt-$\mathcal{SL}$. The third one, p_4 is discarded, as it is dominated by both the node $\{p_1\}^{(1)}$ and p_3 in the node $\{p_2, p_3\}^{(2)}$. Next, the leaf node $[p_5, p_7]$ is explored. Since all the enclosed data points are not dominated by two or more other data points, they are appended to cnt-$\mathcal{SL}$. Last, the leaf node $[p_8, p_9]$ is examined against cnt-$\mathcal{SL}$ and discarded since it is dominated by p_1 and p_3. The stack becomes empty. The search terminates and the skyband query result is $\{p_1, p_2, p_3, p_5, p_6, p_7\}$. □

Other approaches such as SFS and BBS (i.e., two of the representative approaches as reviewed in Sect. 2) can be intuitively extended to answer skyband queries. However, they are not efficient. This is because SFS does not cluster data points and it is forced to compare data points individually, and BBS, which keeps dominance regions in the main-memory R-tree, cannot efficiently determine the number of data points that dominate a given index node and/or data point. Because of the nature of R-tree, a data point covered by the MBR associated with an internal node might not be covered by some of its enclosed skyline candidates' dominance regions (see Sect. 2.2). Hence, maintaining *counts* of underlying dominance regions in intermediate nodes in main-memory R-tree cannot provide the same effect as cnt-$\mathcal{SL}$ and thereafter, all traversals have to reach the leaf levels in order to determine the number of dominating data points.

6.3 The *ZRank* algorithm

In this subsection, we present the *ZRank* algorithm that can efficiently answer top-ranked skyline queries. Its efficiency is attributed to (i) an efficient dominance test mechanism integrated with counting of dominated data points, (ii) cnt-$\mathcal{SRC}$ that indexes a source dataset and provides cnts of data points associated with index nodes, and (iii) dom-$\mathcal{SL}$ that indexes skyline candidates plus doms associated with individual index nodes and candidates indicating the number of data points they dominate. These cnt-$\mathcal{SRC}$ and dom-$\mathcal{SL}$ are *extended* ZBtrees. As briefly described, the *ZRank*

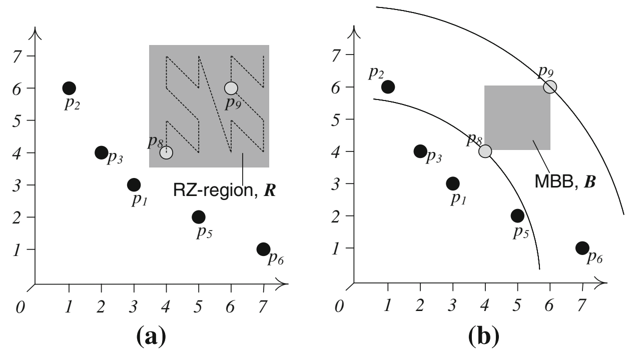


Fig. 20 An example for top-ranked skyline queries. **a** dominance and counting, **b** counting based on BBS

algorithm operates in two phases, i.e., *counting* phase and *ranking* phase. The counting phase searches for skyline candidates and counts the numbers of their dominated data points. The ranking phase orders the skyline candidates according to their dominating power and returns t candidates with the largest dominating power as the result (where t is controlled by a user). We detail these two phases below.

The counting phase. For conventional skyline queries, dominance tests simply determine whether a data point is dominated. Once a node is dominated, the whole branch rooted by the node in $\mathcal{SRC}$ is skipped from further examinations. However, for top-ranked skyline queries, skipping detailed examinations of some dominated nodes would result in incorrect counts. Figure 20a illustrates a case in which skipping exploring an RZ-region leads to an incorrect counting. Recall that the skyline points are $\{p_1, p_2, p_3, p_5, p_6\}$ in our running example and an RZ-region R that contains p_8 and p_9 is completely dominated by p_1. However, p_9 (not p_8) in R is dominated by p_2 and p_5 and so it should contribute to the dominating powers of both p_2 and p_5. If R is not further explored once it is found to be dominated by p_1, the counts of p_2 and p_5 would miss p_9, i.e., incorrect counts received by p_2 and p_5. As such, nodes cannot simply be discarded as they may have enclosed data points dominated by other skyline candidates.

We derive *DominateAndCount* function that incorporate the counting of dominated points for skyline points with dominance tests. The basic idea of the *DominateAndCount* function is that when comparing an examinee RZ-region that bounds one or multiple data points against existing skyline points in dom-$\mathcal{SL}$, the counting is performed simultaneously. To facilitate the counting, each node n in dom-$\mathcal{SL}$ is associated with a counter dom that records the number of data points dominated by n. Whenever n is found to dominate some x points, its dom (as initialized to 0) is incremented by x. We assume an extended ZBtree in which the number of data points cnt beneath very node n is available at n. Thus, in the function, an examinee RZ-region R from cnt-$\mathcal{SRC}$ is

```
Function $DominateAndCount($
         $dom\text{-}\mathcal{SL}$, $minpt$, $maxpt$, $cnt$, $pminpt$)
Input.   $dom\text{-}\mathcal{SL}$: a ZBtree that indexes skyline points;
         $minpt$ and $maxpt$: the bounds of an RZ-region;
         $cnt$: the count of data points inside RZ-region;
         $pminpt$: the lower endpoint of a parent RZ-region;
Local.   $q$: Queue; $cnt$: a counter of dominating points
Output.  $needToExplore$: a flag: (default: FALSE)
           TRUE if examinee is needed to explore, else FALSE;
         $dominated$: a flag: (default: FALSE)
           TRUE if examinee is dominated, else FALSE;
Begin
 1.  $q$.enqueue($dom\text{-}\mathcal{SL}$'s root);
 2.  while ($q$ is not empty) do
 3.    var $n$: Node;
 4.    $q$.dequeue($n$);
 5.    if ($n$ is a non-leaf node) then
 6.      forall (child node $c$ of $n$) do
 7.        if ($c$'s RZ-region's $maxpt \not\vdash pminpt$) then
 8.          if ($c$'s RZ-region's $maxpt \vdash minpt$) then
 9.            $c$'s $dom \leftarrow c$'s $dom + cnt$; /* Case 1 of Lemma 1 */
10.            $dominated \leftarrow$ TRUE;
11.          else if ($c$'s RZ-region's $minpt \vdash minpt$) then
12.            $q$.enqueue($c$);                  /* Case 2 of Lemma 1 */
13.    else /* leaf node */
14.      forall (child point $c$ of $n$) do
15.        if ($c \not\vdash pminpt$) then      /* avoid redundant count */
16.          if ($c \vdash minpt$) then
17.            $c$'s $dom \leftarrow c$'s $dom + cnt$;
18.            $dominated \leftarrow$ TRUE;
19.          else ($c \vdash maxpt$) then
20.            $needToExplore \leftarrow$ TRUE;
21.  output $needToExplore$, $dominated$;
End.
```

Fig. 21 The pseudo-code of the *DominateAndCount* function

compared with an RZ-region R' from $dom\text{-}\mathcal{SL}$. There are three possible cases. First, R is dominated by R' and hence R's cnt contributes to R''s dom. Second, R may be dominated by some, but not all, data points of R' and a further examination of the examinee is triggered. Third, R is not dominated by R' and the examination on R' is finished.

Since a node n from $cnt\text{-}\mathcal{SRC}$ cannot be discarded even if it is dominated by some skyline candidates, the child entries of n may be further investigated that triggers the traversal of the same nodes in a $dom\text{-}\mathcal{SL}$ again. This raises a *duplicate counting problem*. To tackle this, we include a parent RZ-region of R in the *DominateAndCount* function to provide an additional checking. Thus, if the parent RZ-region is already dominated by R' (that implies R's cnt already included by R''s dom), the further investigation of the examinee can be skipped.

The pseudo-code of the *DominateAndCount* function is listed in Fig. 21. It takes four parameters: (i) a $dom\text{-}\mathcal{SL}$, (ii) an examinee RZ-region expressed as ($minpt$, $maxpt$) from $cnt\text{-}\mathcal{SRC}$, (iii) the number of data points inside the examinee (cnt), and (iv) the examinee's parent RZ-region as the inputs. Since the parent RZ-region R is only used to check if it is completely dominated by existing skyline candidates, we only need $minpt(R)$ (i.e., $pminpt$ in the algorithm). In the function, we maintain a queue q to perform breadth-first traversal in $dom\text{-}\mathcal{SL}$ and a flag $needToExplore$ to indicate the need to explore the examinee. For every dequeued entry, we check its child c against $pminpt$ to avoid redundant counting (line 7 and line 15). Then we increment dom by cnt if the examinee is completely dominated (lines 8–10 and lines 16–18). Otherwise, we further explore the child node c (lines 11–12) or indicate the need to explore the examinee if the leaf level is reached and a part of examinee node is found to be dominated (lines 19–20). For brevity, we omit the two cases in which the examinee is not dominated (as implied by previous two conditions) as stated in Lemma 1. Meanwhile, we maintain a flag $dominated$ to record if an examinee is dominated or not. Finally $needToExplore$ and $dominated$ are returned.

The counting phase utilizes the monotonic ordering property of Z-order curves by which the accuracy of dom is guaranteed as *all* the dominating data points are accessed before an examinee. Therefore, examinees only contribute their counts to those skyline candidates retrieved earlier than them, but definitely not those accessed later. However, other approaches such as BBS cannot achieve this. Accessed in accordance with the shortest distances to the origin, the MBB B is examined before p_2 as shown in Fig. 20b. In this case, BBS does not anticipate some data points inside B (e.g., p_9) to be dominated by some skyline candidates (e.g., p_2) retrieved later. To remedy this problem, BBS has to explore all the index nodes, or to separate the retrieval of skyline candidates and the counting. Obviously, both approaches are inefficient. The former degenerates BBS to a scan-based approach and forces it to examine all individual data points; while the latter scans the index twice. Besides, SFS suffers from exhaustive point-to-point comparisons.

The ranking phase. To rank skyline candidates based on the numbers of data points they dominate, the *Rank* function consolidates skyline candidates' doms by aggregating doms of nodes to the data points at the bottom in $dom\text{-}\mathcal{SL}$. Figure 22 depicts the pseudo-code of the Rank algorithm. It traverses the $dom\text{-}\mathcal{SL}$ based on a depth-first traversal (lines 2–11) and performs sorting on all skyline candidates (line 12). Finally the initial t skyline points with the smallest doms in the sorted list are returned.

Putting all together. Our *ZRank* algorithm (as outlined in Fig. 23) is devised based on both *DominateAndCount* and *Rank* functions. It iteratively examines nodes and data points in non-decreasing order of Z-addresses using a stack to keep track of unexamined entries (lines 2–14). For non-leaf nodes, it examines child nodes c with the *DominateAndCount* function. If it needs to be explored, c is pending in the stack (lines 11–14). For leaf nodes, all child points are examined against those in $dom\text{-}\mathcal{SL}$ with the same function. Those data points not dominated are inserted to $dom\text{-}\mathcal{SL}$. After all entries are examined, the skyline candidates in $dom\text{-}\mathcal{SL}$ are ranked by the *Rank* function and the top t ones are returned. Example 3

```
Function $Rank(dom\text{-}\mathcal{SL}, t)$
Input.   $dom\text{-}\mathcal{SL}$: a ZBtree that indexes skyline points;
         $t$: the requested number of top-ranked skyline points.
Local.   $s$: stack; $l$: result list;
Output.  $t$ top-ranked skyline points
Begin
 1.  $s$.push($dom\text{-}\mathcal{SL}$'s root);
 2.  while ($s$ is not empty) do
 3.    var $n$: Node;
 4.    $s$.pop($n$);    /* depth-first traversal */
 5.    if ($n$ is a non-leaf node) then
 6.      forall (child node $c$ of $n$) do
 7.        $c$'s $dom \leftarrow c$'s $dom + n$'s $dom$;
 8.    else
 9.      forall (child point $c$ of $n$) do
10.        $c$'s $dom \leftarrow c$'s $dom + n$'s $dom$;
11.        insert $c$ to $l$;
12.  sort $l$ in non-descending order of $dom$;
13.  output first $t$ points from $l$;
End.
```

Fig. 22 The pseudo-code of the *Rank* function

```
Algorithm $ZRank(cnt\text{-}\mathcal{SRC}, t)$
Input.   $cnt\text{-}\mathcal{SRC}$: a ZBtree that indexes skyline points;
         $t$: the requested number of top-ranked skyline points.
Local.   $s$: stack; $dom\text{-}\mathcal{SL}$
Output.  $U_t$: $t$ top-ranked skyline points
Begin
 1.  $s$.push($dom\text{-}\mathcal{SL}$'s root);
 2.  while ($s$ is not empty) do
 3.    var $n$: Node;
 4.    $s$.pop($n$);                    /* depth-first traversal */
 5.    if ($n$ is a non-leaf node) then
 6.      forall (child node $c$ of $n$) do
 7.        $needToExplore, dominated \leftarrow$
           DominateAndCount($dom\text{-}\mathcal{SL}$, $c$, $c$, 1, $n$'s $minpt$);
 8.        if (not dominated) then
 9.          $dom\text{-}\mathcal{SL}$.insert($c$);
10.    else
11.      forall (child point $c$ of $n$) do
12.        $needToExplore, dominated \leftarrow$
           DominateAndCount($dom\text{-}\mathcal{SL}$, $c$'s $minpt$,
             $c$'s $maxpt$, $c$'s $cnt$, $n$'s $minpt$);
13.        if ($needToExplore$) then
14.          $s$.push($c$);
15.  $U_t \leftarrow$ Rank($dom\text{-}\mathcal{SL}$,$t$);
16.  output $U_t$;
End.
```

Fig. 23 The pseudo-code of the *ZRank* algorithm

illustrates how the *ZRank* algorithm in invoked to evaluate a top-ranked skyline query.

Example 3 In this example, a top-ranked skyline query with t set to 2 is evaluated on our example dataset. The trace is depicted in Fig. 24. Here, we denote every cnt-$\mathcal{SRC}$ node by $[p, q]^{[cnt]}$ where p and q are the ending data points and cnt is the count of enclosed data points, and denote a dom-$\mathcal{SL}$ node n by $\{p_1^{(dom_1)}, p_2^{(dom_2)}, \cdots\}^{(dom_n)}$ where p_i is a data point enclosed by n, dom_i records the number of data points dominated by p_i, and dom_n records the number of data points n dominate (i.e., all data points enclosed in the corresponding RZ-region). The total number of data points dominated by p_i is therefore $dom_i + dom_n$.

Stack	Skyline points cnt-$\mathcal{SL}$
$[p_1, p_4]^{[4]}$,$[p_5, p_9]^{[5]}$	$\emptyset$
$[p_1, p_1]^{[1]}$,$[p_2, p_4]^{[3]}$, $[p_5, p_9]^{[5]}$	$\emptyset$
$[p_2, p_4]^{[1]}$,$[p_5, p_9]^{[5]}$	$\{p_1^{(0)}\}^{(1)}$
$[p_5, p_9]^{[5]}$	$\{p_1^{(0)}\}^{(1)}$,$\{p_2^{(1)}, p_3^{(1)}\}^{(0)}$
$[p_5, p_7]^{[3]}$,$[p_8, p_9]^{[2]}$	$\{p_1^{(0)}\}^{(1)}$,$\{p_2^{(1)}, p_3^{(1)}\}^{(0)}$
$[p_8, p_9]^{[2]}$	$\{p_1^{(0)}\}^{(3)}$,$\{p_2^{(1)}, p_3^{(3)}\}^{(0)}$, $\{p_5^{(1)}, p_6^{(0)}\}^{(0)}$
$p_8^{[1]}$,$p_9^{[1]}$	$\{p_1^{(0)}\}^{(3)}$,$\{p_2^{(1)}, p_3^{(3)}\}^{(0)}$, $\{p_5^{(1)}, p_6^{(0)}\}^{(0)}$
$p_9^{[1]}$	$\{p_1^{(0)}\}^{(3)}$,$\{p_2^{(2)}, p_3^{(3)}\}^{(0)}$, $\{p_5^{(2)}, p_6^{(0)}\}^{(0)}$

Fig. 24 The trace of the *ZRank* algorithm

First, a stack that keeps unexplored cnt-$\mathcal{SRC}$ nodes is set with the root of cnt-$\mathcal{SRC}$. Next, the search explores the first branch $[p_1, p_4]^{[4]}$, and then drills down the first leaf node $[p_1, p_1]^{[1]}$ where a data point p_1 is collected as the first skyline point. Thereafter, the second leaf node $[p_2, p_4]^{[3]}$ is examined and explored. The enclosed data points p_2 and p_3 are collected as the second and third skyline points in dom-$\mathcal{SL}$. Then, p_4 is evaluated and is dominated by an entire leaf node $\{p_1^{(0)}\}$, and two data points p_2 and p_3. All corresponding counts are incremented by one. Notice that p_4, although dominated by both p_2 and p_3, is not dominated by the node $[p_2, p_3]$ and hence its cnt contributes to the doms of p_2 and p_3, but not that of node $[p_2, p_3]$. Further, $[p_5, p_9]^{[5]}$ is explored. Since it is incompatible to $\{p_2^{(1)}, p_3^{(1)}\}^{(0)}$ and not dominated by $\{p_1^{(0)}\}$, $[p_5, p_7]^{[3]}$ is explored. Then, p_5 and p_6 are collected as the fourth and fifth skyline points maintained as $\{p_5^{(0)}, p_6^{(0)}\}^{(0)}$ in dom-$\mathcal{SL}$. As p_7 is only dominated by p_5, its count contributes to p_5's dom.

Next, the leaf node $[p_8, p_9]^{[2]}$ is completely dominated by $\{p_1^{(0)}\}$ and p_3, so both the doms of $\{p_1^{(0)}\}$ and p_3 are increased to 3. At the same time, it may contain data points dominated by p_2, p_5 and p_6. Thereafter, $[p_8, p_9]^{[2]}$ is further explored. When p_8 is examined, as the associated parent $minpt$, i.e, p_8 is already dominated by all the dominating points, no count is updated. As for p_9, it is dominated by p_2 and p_5 and hence it increases both p_2's and p_5's doms by one. After the entire cnt-$\mathcal{SRC}$ is traversed, the counting phase completes and the ranking phase starts. The counts of intermediate nodes in dom-$\mathcal{SL}$ are propagated to the data points so the counts for skyline points, p_1, p_2, p_3, p_4 and p_5, become 3, 2, 3, 2 and 0, respectively. Data points p_1 and p_3 are picked as the query result to complete the search. $\square$

6.4 The *k-ZSearch* algorithm

Next, we present the *k-ZSearch* algorithm to evaluate k-dominant skyline queries. To address the *cyclic dominance* problem, our *k-ZSearch* algorithm adopts a filter-and-reexamine approach. In the filtering phase, possible k-dominant skyline candidates, which may contain false hits are collected. The reexamination phase eliminates those false hits. We detail these two phases in the following.

The filtering phase. In the filtering phase, the *k-ZSearch* algorithm traverses $\mathcal{SRC}$ for k-dominant skyline candidates while filtering out those data points or nodes that are k-dominated by collected skyline candidates. Here, candidates are maintained in k-$\mathcal{SL}$ indexed by a *ZBtree*. Extended from the definition of the dominance relationship to the k-dominance relationship between RZ-regions, Theorem 1 formalizes the transitive k-dominant relationship; and Lemma 2 suggests an efficient candidate filtering in the *k-ZSearch* algorithm.

Theorem 1 Transitive k-dominance relationship. *The following two transitive k-dominance relationships are true:*

1. *If $p \vdash_k p'$ and $p' \vdash p''$, then $p \vdash_k p''$.*
2. *If $p \vdash p'$ and $p' \vdash_k p''$, then $p \vdash_k p''$.*

Proof For (1), given certain k out of d dimensions, s_i, $p \cdot s_i \leq p' \cdot s_i \leq p'' \cdot s_i$ (where $s_i \in S' \wedge S' \subseteq S \wedge |S'| = k$) must hold, implying $p \cdot s_i \leq p'' \cdot s_i$. Suppose there is one dimension out of k dimensions, s_j, such that $p \cdot s_j < p' \cdot s_j \leq p'' \cdot s_j$, in this case, $p \cdot s_j < p'' \cdot s_j$. Hence, $p \vdash_k p''$.

Likewise for (2), given certain k out of d dimensions, s_i, $p.s_i \leq p'.s_i \leq p''.s_i$ (where $s_i \in S' \wedge S' \subseteq S \wedge |S'| = k$) must hold, so $p.s_i \leq p''.s_i$. Also, there exists at least one dimension s_j among those k dimensions, $p.s_j \leq p'.s_j < p''.s_j$, then $p.s_j < p''.s_j$. Thus, $p \vdash_k p''$. □

Lemma 2 *Given two RZ-regions, R and R', the following four cases hold:*

1. *if $maxpt(R) \vdash_k minpt(R')$, then $R \vdash_k R'$, then any data point in R k-dominates all the data points in R'.*
2. *if $maxpt(R) \not\vdash_k minpt(R') \wedge minpt(R) \vdash_k minpt(R')$, then some data points in R may k-dominate all the data points in R.*
3. *if $minpt(R) \not\vdash_k minpt(R') \wedge minpt(R) \vdash_k maxpt(R')$, then some data points in R would k-dominate some data points in R'.*
4. *if $minpt(R) \not\vdash_k maxpt(R')$, then $R \not\vdash_k R'$, then no data point in R can k-dominate any data point in R'.*

Proof We prove the lemma case by case.

Case 1. $maxpt(R) \vdash_k minpt(R')$. According to the first part of Theorem 1, since $maxpt(R) \vdash_k minpt(R')$, $minpt(R')$ k-dominates all the data points in R'. By the second part of Theorem 1, all the data points in R (except $maxpt(R)$) dominate $maxpt(R)$. As a result, $R \vdash_k R'$. As shown in Fig. 25a, R_1 and R_2 are 2-dominated by R_s in a 3D space.

Case 2. $maxpt(R) \not\vdash_k minpt(R') \wedge minpt(R) \vdash_k minpt(R')$. Some data points in R, such as $minpt(R)$, that can k-dominate $minpt(R')$. Based on the first part of Theorem 1, they can also k-dominate all the other points in R'. In Fig. 25a, R_s and R_3 shows an example.

Case 3. $minpt(R) \not\vdash_k minpt(R') \wedge minpt(R) \vdash_k maxpt(R')$. Some data points in R' including $maxpt(R')$ might be k-dominated by some points in R. R_s and R_4 in Fig. 25a form an example.

Case 4. $minpt(R) \not\vdash_k maxpt(R')$. The case can be proved by contradiction. Suppose $p \vdash_k p'$ ($p \in R$, $p' \in R'$). By Theorem 1, $minpt(R) \vdash p \vdash_k p' \vdash maxpt(R') \Rightarrow minpt(R) \vdash_k maxpt(R')$. This contradicts the case condition. R_s and R_5 in Fig. 25a are an example for this case. □

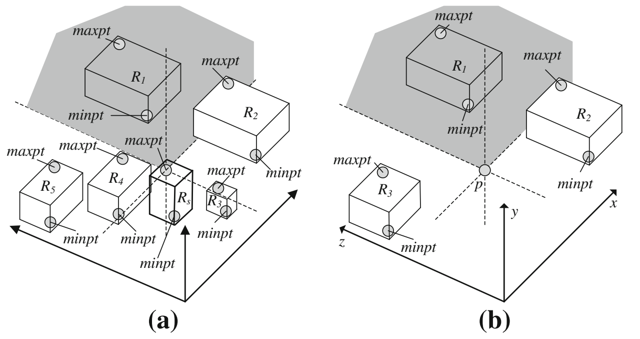


Fig. 25 2-Dominance tests in 3D space. **a** RZ-regions, **b** a point and RZ-regions

```
Function k-Dominate(k-SL, minpt, maxpt)
Input.   k-SL: a ZBtree (k-dominant skyline candidates);
         minpt and maxpt: the endpoints of an RZ-region;
Local.   q: Queue;
Output.  TRUE if the input is k-dominated, else FALSE;
Begin
  1.  q.enqueue(k-SL's root);
  2.  while (q is not empty) do
  3.    var n: Node;
  4.    q.dequeue(n);
  5.    if (n is a non-leaf node) then
  6.      forall (child node c of n) do
  7.        if (c's maxpt ⊢_k minpt) then
  8.          output TRUE;        /* Case 1 of Lemma 2 */
  9.        else if (c's mint ⊢_k maxpt) then
 10.          q.enqueue(c);       /* Case 2 of Lemma 2 */
 11.    else                              /* n is leaf node */
 12.      forall (child point c of n) do
 13.        if (c ⊢_k minpt) then
 14.          output TRUE;
 15.  output FALSE;
End.
```

Fig. 26 The pseudo-code of the *k-Dominate* function

Lemma 2 is applicable to data points (see Fig. 25b). Based on this, we derive the *k-Dominate* function to determine whether an RZ-region, R, presented as $minpt$ and $maxpt$, is k-dominated by any existing candidate k-dominant skyline points preserved in k-$\mathcal{SL}$. The logic of this function is outlined in Fig. 26. Similar to the *Dominate* function, the *k-Dominate* terminates when R is assured to be k-dominated

```
Algorithm $k$-$ZSearch(\mathcal{SRC})$
Input.   $\mathcal{SRC}$: a ZBtree for source data set;
Local.   $k$-$\mathcal{SL}$: a ZBtree ($k$-dominance skyline candidates);
         $s$: Stack; $T$: Set;
Begin
      /*** filtering phase ***/
 1.   $s$.push($source$'s root);
 2.   while $s$ is not empty do
 3.     var $n$: Node;
 4.     $s$.pop($n$);
 5.     if (Dominate($k$-$\mathcal{SL}$, $n$'s $minpt$, $n$'s $maxpt$)) then
 6.       goto line 3. /* remove those dominated data points */
 7.     if ($k$-Dominate($k$-$\mathcal{SL}$, $n$'s $minpt$, $n$'s $maxpt$)) then
 8.       $T$.insert($n$);                    /* $n$ for reexamination. */
 9.       goto line 3.
10.     if ($n$ is a non-leaf node) then
11.       forall (child node $c$ of $n$) do
12.         $s$.push($c$);
13.     else /* leaf node */
14.       forall (child point $c$ of $n$) do
15.         if ($k$-Dominate($k$-$\mathcal{SL}$, $c$, $c$)) then
16.           $T$.insert($c$);                /* $c$ for reexamination. */
17.         else
18.           $k$-$\mathcal{SL}$.insert($c$);          /* tentative result set */
      /*** reexamination phase ***/
19.   forall (point $p \in k$-$\mathcal{SL}$) do
20.     if ($\exists p' \in k$-$\mathcal{SL}$, $p' \neq p \wedge p' \vdash_k p$) then
21.       remove $p$ from $k$-$\mathcal{SL}$;
22.       $T$.insert($p$);
23.     else if ($\exists p'' \in T$, $p'' \vdash_k p$) then
24.       remove $p$ from $k$-$\mathcal{SL}$;
25.       $T$.insert($p$);
26.   output $k$-$\mathcal{SL}$;
End.
```

Fig. 27 The pseudo-code of the *k-ZSearch* algorithm

by any RZ-region enclosing some candidate skyline points, or when all relevant nodes of k-$\mathcal{SL}$ are visited.

With the *k-Dominate* function, the filtering phase of the *k-ZSearch* algorithm (lines 1–18 in Fig. 27) collects k-dominant skyline candidates in k-$\mathcal{SL}$. Those unqualified data points or index node entries (i.e., k-dominated by any existing result candidate) are reserved in a non-candidate set (T) which will be used in the reexamination phase for false hit removal. The memory consumption for T is expected to be low, because most of branches of $\mathcal{SRC}$ can be pruned at high levels owing to the k-dominance relationship. The filtering phase terminates when the $\mathcal{SRC}$ is completely traversed.

The reexamination phase. The reexamination phase of the *k-ZSearch* algorithm removes the false hits from candidates collected in the filtering phase. The main idea is that if a candidate in k-$\mathcal{SL}$ is found to be k-dominated by any other point in k-$\mathcal{SL}$ or in T, it is moved from k-$\mathcal{SL}$ to T. All candidate points are reexamined. The logic of this phase is depicted in lines 19–26 in Fig. 27.

As the number of candidates maintained in k-$\mathcal{SL}$ is much smaller than that of points stored in T and those candidates usually have strong dominating power, our reexamination checks every candidate p against others in k-$\mathcal{SL}$ first. If there is a candidate p' k-dominating p, p is moved to T. Otherwise, we proceed to check p against entries in T. If no nodes and/or

```
Algorithm $ZSubspace(\mathcal{SRC}, S')$
Input.   $\mathcal{SRC}$: ZBtree for source data set;
         $S'$: a subset of dimensions;
Local.   $S'$-$\mathcal{SL}$: ZBtree (subspace skyline points);
         $s$: Stack;
Begin
      /*** filtering phase ***/
 1.   $s$.push($source$'s root);
 2.   while ($s$ is not empty) do
 3.     var $n$: Node;
 4.     $s$.pop($n$);
 5.     if ($Subspace$-$Dominate$($S'$-$\mathcal{SL}$, $n$'s $minpt$, $n$'s $maxpt$)) then
 6.       goto line 3.          /* remove those dominated data points */
 7.     if ($n$ is a non-leaf node) then
 8.       forall (child node $c$ of $n$) do
 9.         $s$.push($c$);
10.     else /* leaf node */
11.       forall (child point $c$ of $n$) do
12.         if (not $Subspace - Dominate$($S'$-$\mathcal{SL}$, $c$, $c$)) then
13.           $S'$-$\mathcal{SL}$.insert($c$);          /* tentative result set */
      /*** reexamination phase ***/
14.   forall (point $p \in S'$-$\mathcal{SL}$) do
15.     if ($Subspace$-$Dominate$($S'$-$\mathcal{SL}$, $p$, $p$)) then
16.       $S'$-$\mathcal{SL}$.delete($p$);
17.   output $S'$-$\mathcal{SL}$;
End.
```

Fig. 28 The pseudo-code of the *ZSubspace* algorithm

data points in T k-dominate p, p retains in k-$\mathcal{SL}$. If some index nodes in T need further exploring, they are replaced with all their child entries (either data points or child nodes) in T. Those final remainders in k-$\mathcal{SL}$ are the k-dominant skyline points.

6.5 The *ZSubsapce* algorithm

At last, we present the *ZSubspace* algorithm to process subspace skyline queries. Since the access of data points based on the ascending order of Z-addresses cannot exploit the transitivity property for subspace skyline queries. Thus, the algorithm also adopts a filter-and-reexamine approach.

In the filtering phase, $\mathcal{SRC}$ is access based on a depth-first traversal that retrieves data points and nodes in the ascending order of Z-addresses. Following this order, it collects candidates if they are not S'-dominated (where S' is a set of dimensions specified at the query time). Theorem 2 suggests the transitive subspace-dominance relationship and Lemma 3 suggests to prune the search space if the RZ-regions of certain nodes are subspace-dominated by current skyline candidates. Thanks to the clustering property of Z-order curves, the *ZSubspace* algorithm can effectively prune the search space in the filtering phase. In Fig. 28, line 1-13 outlines the filtering phase. While subspace dominance relationship is used in place of conventional dominance relationship, the logic of *Subspace-Dominate* function is very similar to the *Dominate* function and we do not include it for space saving.

Theorem 2 Transitive subspace dominance relationship. *Given a subset of dimensions S', the following two transitive subspace dominance relationships are true:*

Stack	S'-$\mathcal{SL}$
$[p_1, p_4], [p_5, p_9]$	$\emptyset$
$[p_1], [p_2, p_4], [p_5, p_9]$	$\emptyset$
$[p_5, p_7], [p_8, p_9]$	$\{p_1\}$
$[p_8, p_9]$	$\{p_1\}, \{p_5, p_6\}$

Fig. 29 The trace of the filter phase of the *ZSubspace* algorithm

1. *If $p \vdash_{S'} p'$ and $p' \vdash p''$, then $p \vdash_{S'} p''$.*
2. *If $p \vdash p'$ and $p' \vdash_{S'} p''$, then $p \vdash_{S'} p''$.*

Proof The proof is very similar to that for Theorem 1. We omit the discussion to save space. □

Lemma 3 *Given a subset of dimensions, S' and two RZ-regions, R and R', the following four cases hold:*

1. *if $maxpt(R) \vdash_{S'} minpt(R')$, then $R \vdash_{S'} R'$, then any data point in R S'-dominates all the data points in R'.*
2. *if $maxpt(R) \nvdash_{S'} minpt(R') \wedge minpt(R) \vdash_{S'} minpt(R')$, then some data points in R may S'-dominate all the data points in R.*
3. *if $minpt(R) \nvdash_{S'} minpt(R') \wedge minpt(R) \vdash_{S'} maxpt(R')$, then some data points in R would S'-dominate some data points in R'.*
4. *if $minpt(R) \nvdash_{S'} maxpt(R')$, then $R \nvdash_{S'} R'$, then no data point in R can S'-dominate any data point in R'.*

Proof This can be proved as that for Lemma 2. Due to limited space, we do not state the proof here. □

In the reexamination phase (as shown in lines 14–16), candidates are reexamined to eliminate false hits. Here, we compare it against all other candidates in S'-$\mathcal{SL}$ only. Those S'-dominated are removed from S'-$\mathcal{SL}$. Finally, the remainders in S'-$\mathcal{SL}$ are the subspace skyline points and the algorithm terminates. To illustrate the operation of *ZSubspace* algorithm, Example 4 is provided below.

Example 4 Consider a subspace skyline query on distance dimension based on our hotel example (i.e., $S' = \{distance\}$). The result is $\{p_6\}$. The trace of the filter phase is shown in Fig. 29. Based on the depth-first traversal on a ZBtree, $\{p_1\}$ is the first data point accessed and it is included in S'-$\mathcal{SL}$. Then the node $[p_2, p_4]$ is S'-dominated by p_1 and ignored. Next, $[p_5, p_7]$ is examined against p_1 and expanded. Thereafter, p_5 is visited and kept in S'-$\mathcal{SL}$. Further, p_6 is not dominated and collected. Later, p_7 and $[p_8, p_9]$ are dominated and pruned. Then the filter phase ends. Notice that p_1 is not a result point but it helps filtering out the nodes $[p_2, p_4]$ and $[p_8, p_9]$.

Next, in the examination phase, p_1 and p_5 are both dominated by p_6 and are removed from S'-$\mathcal{SL}$. Now, p_6, retained in S'-$\mathcal{SL}$ is the subspace skyline point.

7 Performance evaluation

This section evaluates the performance of the proposed suite of algorithms in our Z-SKY framework, namely, *ZSearch*, *ZInsert*, *ZDelete*, *ZUpdate*, *ZBand*, *ZRank*, *k-ZSearch* and *ZSubspace* and compares them with the state-of-the-art approaches specialized for corresponding domains.

7.1 Experiment settings

Our evaluations are based on both synthetic and real datasets. Synthetic datasets are generated and they follow *correlated* distribution, *independent* distribution and *anti-correlated* distribution as discussed in [4] with various data dimensionalities (d) and cardinalities (n). Due to the limited space, we present the results for d ranging from 4, 8, 12 and 16 and n from 10k to 10, 000k (ten millions) in order to demonstrate the scalability of the proposed algorithms. In addition, three real datasets, i.e., NBA, HOU and FUEL that follow anti-correlated, independent and correlated distributions, respectively,[9] are employed. NBA contains 17k 13-dimensional data points, each of which corresponds to the statistics of an NBA player's performance in 13 aspects (such as points scored, rebounds, assists, etc.). HOU consists of 127k 6-dimensional data points, each representing the percentage of an American family's annual expense on 6 types of expenditures such as electricity, gas, and so on. FUEL is a 24k 6-dimensional dataset, in which each point stands for the performance of a vehicle (such as mileage per gallon of gasoline in city and highway, etc.). In the experiments, all datasets are normalized to $[0, 1024)^d$.

Besides our proposed algorithms, we include the state-of-the-art algorithms, namely, SFS [8], SaLSa [2], BBS [22], BBS-Update [22], DeltaSky [31], SFSBand, SFSRank, BBS-Band, BBSRank, TSA [5] and SUBSKY [29] for comparison.[10] SFSBand and BBSBand are devised based on SFS and BBS, respectively, to answer skyband queries. SFSRank and BBSRank are developed based on SFS and BBS, respectively to answer top-ranked skyline queries. Similar to most of the related works in the literature, we use *elapsed time* as the main performance metric. It represents the duration from the time an algorithm starts to the time the result is completely returned. To facilitate our understandings of how elapsed times are spent, we also report the time spent on performing *dominance test*, *data access* and *maintaining skyline candidates* (or candidate maintenance), i.e., the three major operations in the algorithms. Here, data access time includes data retrieval and traversing the required indices.

[9] Those are collected from http://www.nba.com, http://www.ipums.org and http://www.fueleconomy.gov, respectively.

[10] Since DeltaSky focuses only on deletion, we adopt the insertion algorithm of BBS-Update for DeltaSky.

Table 2 Experiment settings

Parameters	Settings (* means default)
Synthetic datasets	Distribution: Correlated, Independent, Anti-correlated
	Dimensionality (d), 4, 8*, 12, 16; Cardinality (n): 10k, 100k, 1000k*, 10, 000k
Real datasets	NBA (13D, 17k, anti-correlated), HOU (6D, 127k, independent), FUEL (6D, 24k, correlated)
Data space	$[0, 1024)^d$
Skyline algorithms	SFS, SaLSa, BBS, *ZSearch*
Skyline update algorithms	BBS-Update, DeltaSky, *ZInsert*, *ZDelete*, *ZUpdate* (number of updates: 30)
Skyband algorithms	SFSBand, BBSBand, *ZBand* (skyband width (b): 2, 4, and 8)
Top-ranked skyline algorithms	SFSRank, BBSRank, *ZRank* (result set size (t): 100)
k-dominant skyline algorithms	TSA and k-*ZSearch* (k: 11, 12, 13*, 14, 15 (where $d = 16$))
Subspace skyline algorithms	SUBSKY and *ZSubspace* (d'=$d/4$, $d/2$ and $d/1$)

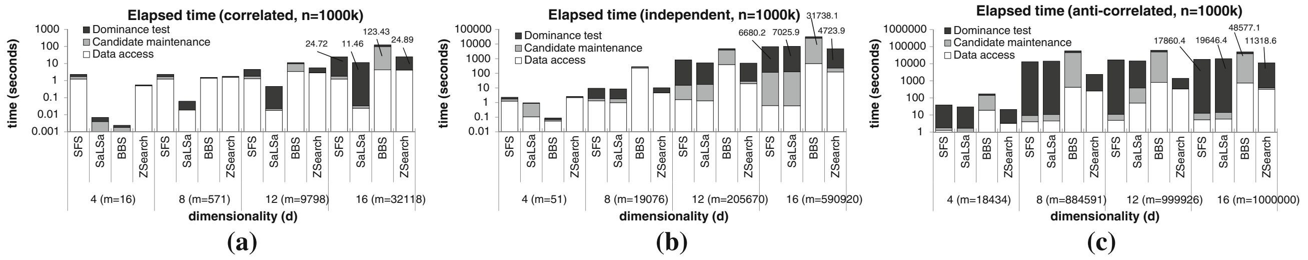


Fig. 30 Skyline query: elapsed time versus dimensionalities (d). **a** correlated, **b** independent, **c** anti-correlated

In addition, we include the *runtime memory consumption* as well as the *number of skyline points* for references.

We implemented all of the evaluated approaches in GNU C++ and conducted the experiments on Linux Servers (running kernel version 2.6.9 with Intel Xeon CPU 3.2 GHz and 4 GB RAM). The disk page size is fixed at 4KB. In the experiments, sufficient memory (including both main memory and virtual memory provided by OS) was available to accommodate all skyline candidates and required data structures. Since Z-addresses can be used to derive the original attribute values, we only keep Z-addresses in *ZBtree*s and *extended ZBtree*s used in the experiments and derive the original dimensional values as needed. Notice that the size of a Z-address equals to the total size of the corresponding original dimensional values. Here, each original value is stored in a two-byte short integer. The R-tree adopted by BBS, BBS-Update, DeltaSky, BBSBand and BBSRank are built with TGS bulkloading [12]. For SFS, TSA, SFSBand and SFS-Rank, records are presorted in accordance with the sum of all attribute values and for SaLSa, records are sorted according to their minimum attribute values. All indices and sorted records are prepared prior to the experiments. $\mathcal{SRC}$s in form of *ZBtree*s and *extended ZBtree*s are built with compact bulkloading (see Sect. 3.3). The results to be reported are the averaged performance of 30 sample runs. Table 2 summarizes all the parameters and their settings used in the experiments. In the following, we first evaluate the performance of skyline search, and then the performance of skyline result update followed by the evaluation of algorithms for skyline variants.

7.2 Experiments on skyline queries

The first experiment set evaluates *ZSearch*, compares it against SFS, SaLSa and BBS with synthetic datasets with various data distributions, dimensionality and cardinalities, and examines its practicality with the real datasets.

Effect of data dimensionality. Figure 30 plots the elapsed time (in log scale) against the data dimensionality (d) from 4 up to 16 in a step of 4 for synthetic correlated, independent and anti-correlated datatsets while data cardinality (n) is fixed at 1,000k. The numbers of skyline points (m) are marked right below the x-axis. The numbers of node accesses (for $d = 16$) are shown above the bars. We notice that all the algorithms incur longer elapsed time with the increase of the data dimensionality. We also find that dominance tests consume more time in anti-correlated datasets than others. SaLSa improves SFS by terminating the search earlier. However, SFS and SaLSa perform inefficient point-to-point dominance tests, resulting in longer elapsed times. BBS performs the best for correlated and independent datasets in low dimensionalities (e.g., $d = 4$), since many branches in R-trees are pruned. However, its performance significantly deteriorates because of the degraded performance of R-trees and the overheads

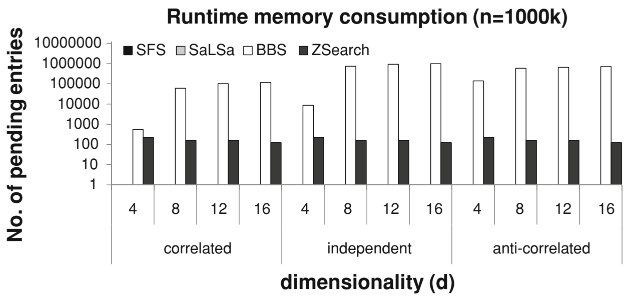


Fig. 31 Skyline query: runtime memory versus dimensionalities (d)

incurred by dominance tests based on main-memory R-trees and by heaps manipulating the pending entries. This is also reflected as the maximum number of pending entries as shown in Fig. 31. Finally, *ZSearch*, owing to the effective space pruning capability and block-based dominance tests, performs better than others when the data dimensionalities increase.

For anti-correlated datasets, data points in some senses are clustered but located in positions in the data space that they do not dominate each other. In this case, *ZSearch* shows its superiority over SFS, SaLSa and BBS. As most of data points cannot be dominated, computational overheads in performing dominance tests become more significant than those under the other two data distributions. We can see that BBS performs the worst due to inefficient dominance tests based on main-memory R-trees and exhaustive heap manipulations for almost entire datasets. Meanwhile, SFS and SaLSa perform worse than *ZSearch* as they cannot avoid many comparisons.

Figure 31 depicts the runtime memory consumption measured in terms of the maximum number of entries maintained in memory to facilitate index traversal (in log scale).[11] Since SFS and SaLSa do not need any extra data structure, it incurs zero runtime memory consumption. BBS uses a heap to order index pages and data points; and *ZSearch* uses a stack. BBS takes (up to four orders of magnitudes) more memory than *ZSearch*. Even worse, each heap entry in BBS that maintains the distance to the origin of the space is slightly larger than a *ZSearch* stack entry (which simply is a pointer to a node).

Effect of data cardinalities. Figure 32 depicts the elapsed time against the data cardinalities (n: 10k up to 10,000k) while d is fixed at 8. The numbers of skyline points (m) are listed right below the x-axis. In general, m increases with the sizes of the experimented datasets. There is an exception that m for correlated datasets drops when datasets are increased from 1000 k to 10,000 k. This is because more data points in a larger dataset would appear close to the origin and they can dominate many other data points.

[11] The data structures to keep track of collected skyline points are not counted.

The elapsed times of all the algorithms increase dramatically as n grows. For correlated datasets, SaLSa performs the best since it can early terminates the searches once (a few number of) skyline points are identified. For the independent datasets, the performances of SFS, SaLSa and *ZSearch* are quite close. Conversely, *ZSearch* produces the shortest elapsed time for anti-correlated dataset. As n increases, the costs incurred by dominance tests become predominant and thus, SFS, SaLSa and BBS perform worse than *ZSearch*.

Next, Fig. 33 plots runtime memory consumption under various data cardinalities. SFS and SaLSa incur no extra data structure and thus they produce zero runtime memory consumption. *ZSearch* consumes reasonably small runtime memory, less than a hundred entries for all the cases. BBS keeps a large number of data points and index nodes, especially for high data cardinality. From this, we can conclude that *ZSearch* is more memory efficient than BBS.

Experiments on real datasets. Here, we evaluate the algorithms on the real datasets. The experiment results are listed in Table 3. *ZSearch* clearly outperforms SFS and BBS and it performs very close to SaLSa among the evaluated algorithms for the elapsed time (in seconds).

The experiment results on synthetic datasets consistently show that *ZSearch* is superior to SFS and BBS, especially when large skyline results are derived. *ZSearch* performs as good as SaLSa for relatively small real datasets. In conclusion, we consider that *ZSearch* is a very good skyline search approach. Due to the limited space, we exclude the results for synthetic correlated datasets hereafter.

7.3 Experiments on skyline result updates

The second set of experiments studies the performance of *ZDelete*, *ZInsert* and *ZUpdate* for skyline result update. We employ BBS-Update [22] and DeltaSky [31] as our comparison candidates. To evaluate the performance for skyline results in presence of insertions and deletions, we first randomly selected 30 skyline points from a skyline result to perform deletion and inserted other 30 data points back to the datasets. Each inserted data point corresponds to a deleted point, with the difference of their attribute values bounded by $[-3, 3]$. We adopt BBS-Update, DeltaSky and *ZDelete* followed by *ZInsert* (labeled as *ZDelete*+*ZInsert*) to perform 30 deletions first and then 30 insertions. In addition, we evaluate the performance of *ZUpdate* that performs insertion before deletion. In these experiments, we vary the data dimensionalities (d) from 4 up to 16 and data cardinalities (n) from 10 k up to 10,000 k.

Figure 34 shows the performance of skyline updates for various data dimensionalities (d: 4 through 16), with a fixed cardinality (1000k) in terms of elapsed time obtained by averaging 30 updates. Both of the initial number of skyline

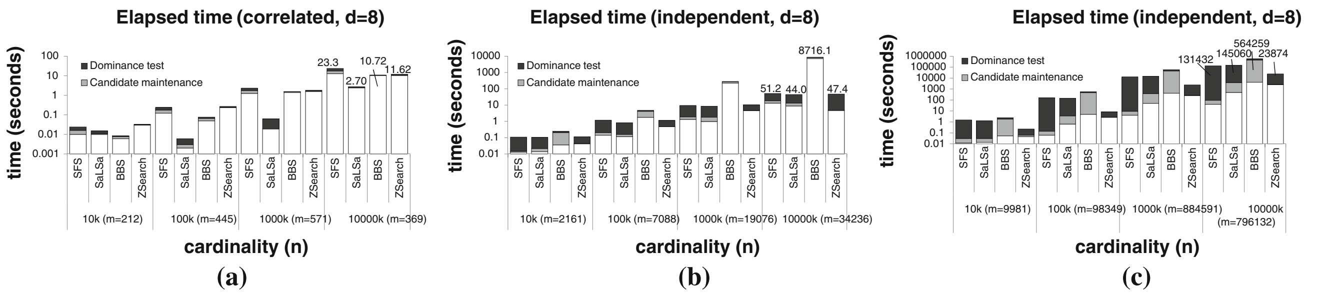


Fig. 32 Skyline query: elapsed time versus cardinalities (n): **a** correlated, **b** independent, **c** anti-correlated

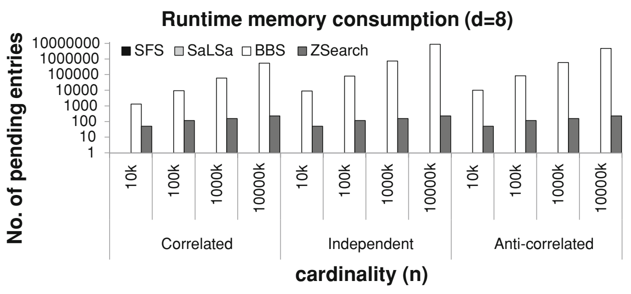


Fig. 33 Skyline query: runtime memory versus cardinalities (n)

Table 3 Skyline query; real datasets (time (s))

Dataset	m	SFS	SaLSa	BBS	*ZSearch*
NBA	10816	2.933	1.702	3.364	1.723
HOU	5774	1.334	0.736	2.169	0.944
FUEL	1	0.031	0.001	0.001	0.001

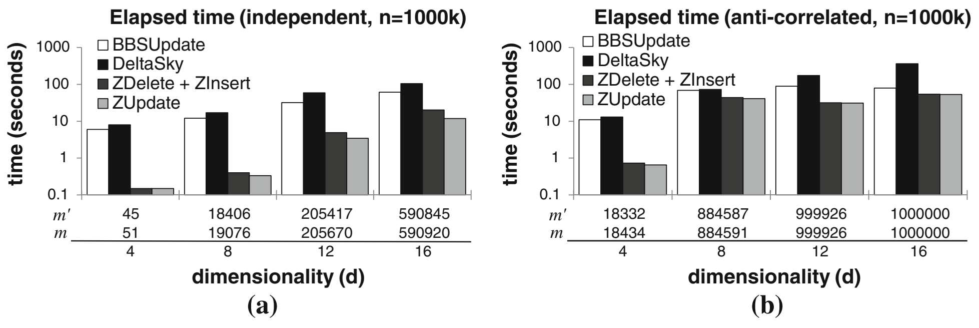
points (m) and the final number of skyline points (m') are stated below the x-axis. As the attribute values of inserted data points are dependent on the values of deleted data points, the differences between m and m', in most cases, are small. As we can observe from the figures, the elapsed times of all the approaches increase when d is increased. This is because a higher dimensionality implies a larger skyline result, which thus incurs more candidate reexaminations. We can also see that BBS-Update and DeltaSky take (almost an order of magnitude) longer elapsed time than both *ZDelete+ZInsert* and *ZUpdate*. This can be explained that they have to examine data points individually against existing skyline points. Besides, the performance of DeltaSky deteriorates as d grows. This is because as d grows, both the length and the number of the sorted lists increase and hence the scanning overhead of lists becomes larger. On the other hand, both *ZDelete+ZInsert* and *ZUpdate* perform batched updates. Further, we can see *ZUpdate* outperforms *ZDelete+ZInsert* since some deleted skyline points are discarded from further examination if they are dominated by inserted data points. Consequently, the corresponding search for those exclusive dominated data points for promotion is eliminated.

Figure 35 shows the performance of all the approaches with the cardinalities varied from 10k to 10000k and the dimensionality fixed at 8. Again, *ZUpdate* is shown to consistently outperform the others due to the aforementioned reasons. Next, we evaluate the update performance on the real datasets. The results are listed in Table 4. The observations are consistent to those obtained from synthetic datasets. Again, *ZUpdate* outperforms *ZDelete+ZInsert* that runs faster than BBS-Update and DeltaSky.

7.4 Experiments on skyband queries

Next, we evaluate *ZBand* for datasets with different data distributions, dimensionality and cardinality in terms of elapsed time, and compare it with SFSBand and BBSBand. Figure 36 plots the elapsed times (in log scale) taken by *ZBand*, SFS-Band and BBSBand upon independent and anti-correlated

Fig. 34 Skyline result update (various dimensionality (d)): **a** independent, **b** anti-correlated



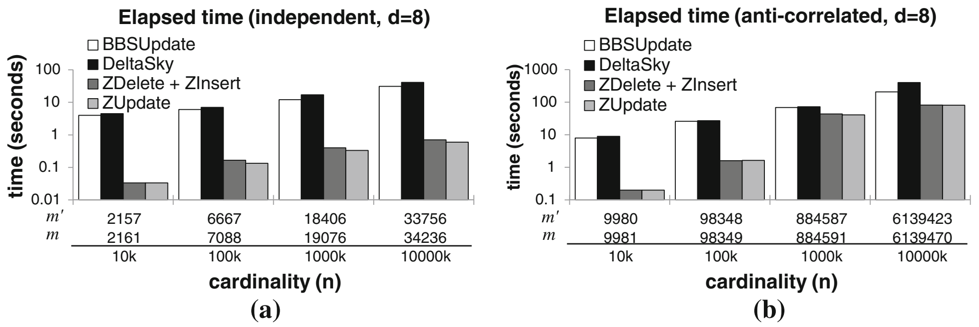


Fig. 35 Skyline result update (various cardinality (n)): **a** independent, **b** anti-correlated

Table 4 Skyline result update: real dataset (time (ms))

Dataset	m/m'	BBS-update	Delta-sky	*ZDelete+ZInsert*	*ZUpdate*
NBA	10816/10801	189	209	58	14
HOU	5774/5734	98	127	27	18
FUEL	1/1	1	1	1	1

datasets with data dimensionality varied from 4 up to 16 and data cardinality fixed at 1000k. Here, the skyband widths (b) are set to 2, 4, and 8. When a larger b is experimented, more skyband points are resulted. In general, *ZBand* performs the best for all the experimented b's, especially when datasets with high dimensionality are experimented. For independent datasets with lower dimensionality (say 4), the number of skyline points (m) is very small. In this case, BBSBand performs better than all the others. However, due to expensive dominance tests that search for b dominating skyband candidates, BBSBand suffers a lot for the rest of experimented datasets.

Figure 37 depicts the experiment results for data cardinality ranged from 10 k up to 10, 000 k while the data dimensionality is fixed at 8. *ZBand* in general outperforms both SFSBand and BBSBand. Further, we evaluate skyband queries on the real datasets. The experiment results are shown in Table 5. For FUEL, which is a correlated dataset and has a small number of skyband points, BBSBand incurs the shortest elapsed time. For NBA and HOU, that are anti-correlated and independent datasets, respectively, *ZBand* performs the best. From these experiment results, we can see that *ZBand* is generally the best skyband query algorithm.

7.5 Experiments on top-ranked skyline queries

Next, we examine the performance of our proposed *ZRank* in comparison with SFSRank and BBSRank for top-ranked skyline queries. Here, the number of returned top-ranked skyline points (t) has a default value of 100. We have conducted experiments to evaluate the performance of different algorithms under various t and found that t does not have any significant impact on the performance. Consequently, the results under different t are skipped. In our implementation of BBSRank, we explore all index nodes to count the number of dominated data points for all individual skyline candidates.

Table 5 Skyband query: real datasets (time (ms))

Dataset	b	m	SFSBand	BBSBand	*ZBand*
NBA	2	12064	2904	5517	1986
	4	12968	3265	6703	2215
	8	13747	3570	7514	2397
HOU	2	7248	1498	4034	1689
	4	8897	2077	5559	2236
	8	10881	2913	7980	3009
FUEL	2	4	57	1	4
	4	5	58	1	3
	8	17	62	1	3

Figure 38 shows the experiment result for datasets with dimensionality varied from 4 up to 16. For all evaluated datasets, *ZRank* is clearly shown to outperform SFSRank and BBSRank. In particular, BBSRank suffers from expensive computational costs in counting dominated data points for all individual skyline candidates. Likewise, SFS also incurs exhaustive dominance tests. Very differently, *ZRank* uses both $cnt\text{-}\mathcal{SL}$ and $dom\text{-}\mathcal{SL}$ to perform block-level dominance tests and counting, making it outperform the others.

Figure 39 shows the experiment results for datasets with cardinality varied from 10 k up to 10, 000 k and dimensionality fixed at 8. Again, for all evaluated datasets, *ZRank* outperforms SFSRank and BBSRank. This can be explained with the above discussed reasons. Further, we evaluate top-ranked skyline queries on the real datasets. From the experiment results as shown in Table 6, we can see that *ZRank* performs much better than SFSRank and BBSRank.

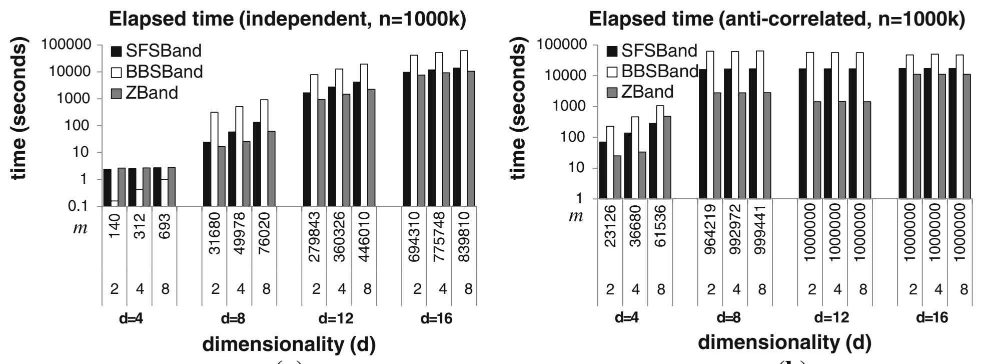


Fig. 36 Skyband query (various dimensionality (d)): **a** independent, **b** anti-correlated

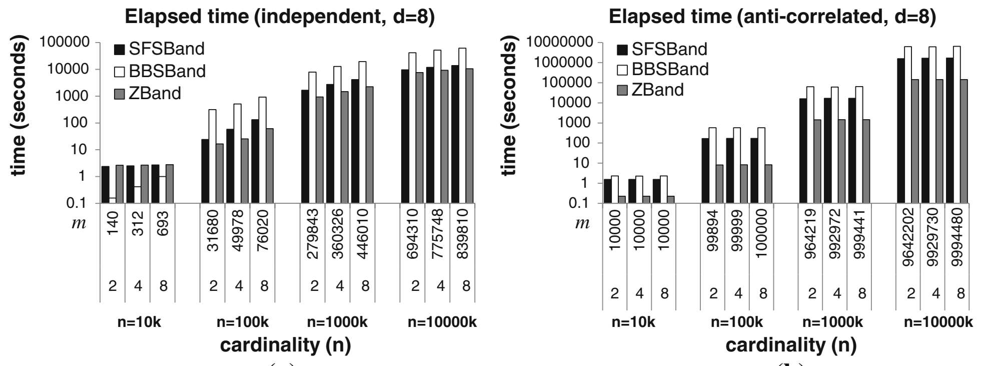


Fig. 37 Skyband query (various cardinality (n)): **a** independent, **b** anti-correlated

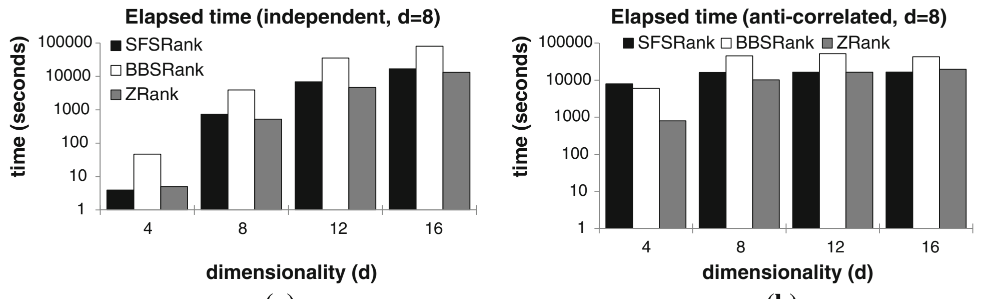


Fig. 38 Top-ranked skyline query (various dimensionality (d)): **a** independent, **b** anti-correlated

Table 6 Top-ranked skyline query: real datasets (time (ms))

Dataset	t	SFSRank	BBSRank	*ZRank*
NBA	100	4206	11561	1095
HOU	100	29121	74006	9332
FUEL	100	56	75	43

7.6 Experiments on k-dominant skyline queries

This set of experiments evaluates k-*ZSearch* for k-dominant skyline queries on high-dimensional datasets. Here, the state-of-the-art approach to be compared with is TSA. The experiment results for various k and a fixed cardinality at 1000 k is shown in Fig. 40. Attributed to block-based dominance tests and a single traversal of *ZBtree*, k-*ZSearch* consistently outperforms TSA as shown in the figure. We can also observe that the number of k-dominant skyline points is reduced when a small k is evaluated, as many data points get k-dominated. The number of skyline candidates is expected to be reduced as well. This results in a shorter processing time. To the other end, the processing time increases as k grows. Due to expensive k-dominance tests to compare more attribute values than the conventional dominance test, the processing times of k-dominance skyline queries are not necessarily shorter than those of skyline queries. Thus, when k approaches d, the number of k-dominant skyline points is close to the number of conventional skyline points, but with longer processing time.

Figure 41 shows the elapsed times of the evaluated algorithms with cardinalities varied from 10 k to 10000 k and k fixed at 13. Again, k-*ZSearch* outperforms TSA and their performance difference becomes more significant as the

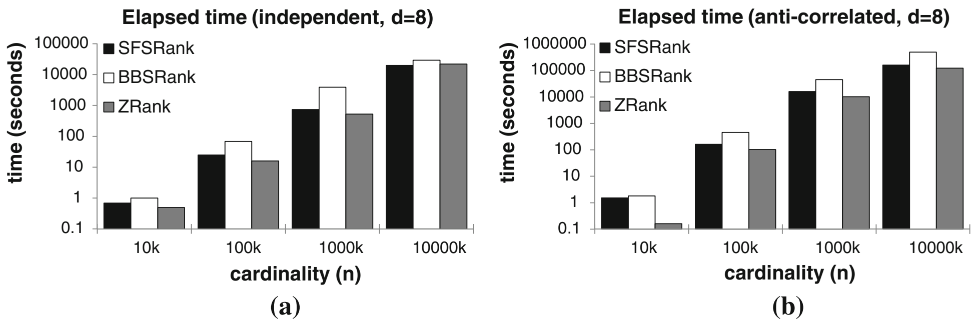


Fig. 39 Top-ranked skyline query (various cardinalities (n)): **a** independent, **b** anti-correlated

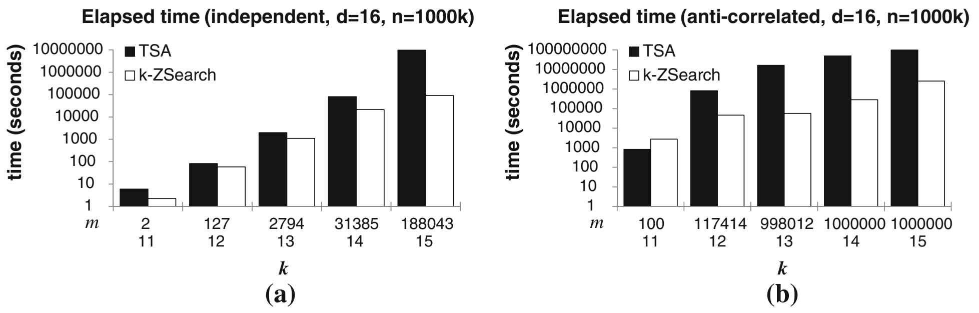


Fig. 40 k-Dominant skyline query (various k): **a** independent, **b** anti-correlated

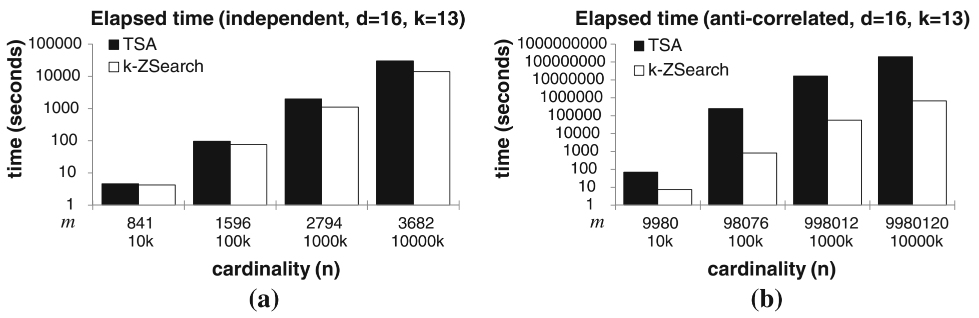


Fig. 41 k-dominant skyline query (various cardinalities (n)): **a** independent, **b** anti-correlated

Table 7 k-Dominant skyline query: real datasets (time (ms))

Dataset	k	m	TSA	*k-ZSearch*
NBA	12	3794	7931	2696
	11	682	1980	731
	10	79	322	171
HOU	5	22	815	226
	4	0	487	220
FUEL	5	1	63	1
	4	1	62	1

cardinality of the dataset increases. This indicates that *k-ZSearch* has a better scalability than TSA.

Finally, we evaluate these algorithms on the real datasets. We set k to $(d-1)$, $(d-2)$, and $(d-3)$ where d represents the dataset dimensionality. Table 7 shows the performance. Consistent with our expectation, *k-ZSearch* performs better than TSA, showing the superiority of *k-ZSearch* for k-dominant skyline queries.

7.7 Experiments on subspace skyline queries

The final set of experiments evaluates the performance of *ZSubspace* and compares it against SUBSKY. We first examine the impact of dimensionalities. We generated synthetic dataset with d set to 4, 8, 12 and 16 and n set to 1000k. With respect to different d, we randomly select $d/4$, $d/2$ and d dimensions as query subspaces. The experimental results in terms of elapsed times are shown in Fig. 42. The numbers of result skyline points are listed right below the x-axis. For independent datasets, SUBSKY performs better than *ZSubspace* when the subspace dimensionalities are small. However, when the subspace dimensionality increases, *ZSubspace* becomes more efficient than SUBSKY due to the effective block-based dominance tests. Due to this reason, *ZSubspace* outperforms SUBSKY for most of the cases in anti-correlated datasets.

Next, we evaluate the performance of both *ZSubspace* and SUBSKY for datasets with cardinalities n varied from 10k to 10000k and dimensionalities set to 8. The results in terms of elapsed times and the number of skyline points are plotted

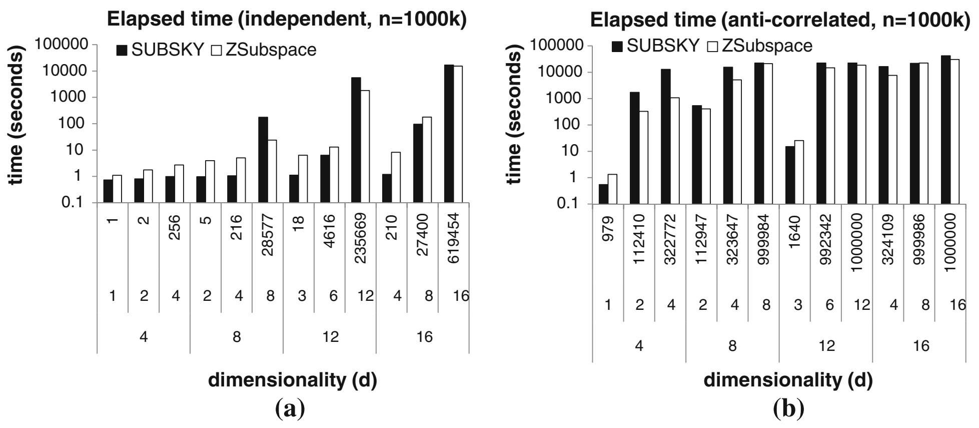


Fig. 42 Subspace skyline query (various dimensionalities (d)): **a** independent, **b** anti-correlated

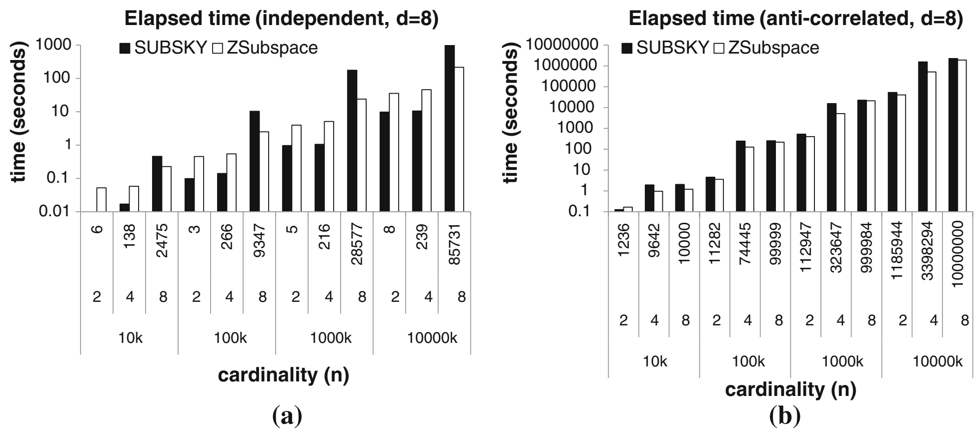


Fig. 43 Subspace skyline query (various cardinalities (n)): **a** independent, **b** anti-correlated

Table 8 Subspace skyline query; real datasets (time (s))

Dataset	$\|S'\|$	m	SUBSKY	*ZSubspace*
NBA	5	397	0.047	0.159
	9	5058	1.217	0.969
	13	10816	4.009	2.225
HOU	2	8	0.049	0.065
	4	294	0.223	0.255
	6	5774	2.121	1.908
FUEL	2	1	0.005	0.003
	4	1	0.005	0.004
	6	1	0.005	0.004

in Fig. 43. For independent datasets, SUBSKY in general performs better than *ZSubspace* except that the number of queried dimensions approaches d. Conversely, for anti-correlated datasets, *ZSubspace* slightly outperforms SUBSKY. This can be explained with the above discussed reasons.

Finally we evaluate both *ZSubspace* and SUBSKY on the real datasets. The results are shown in Table 8. Due to space limitation, we only show some sample numbers of queried dimensions (i.e., $|S'|$). As we can observe from the results, the performance of these two approaches is very close, different from the synthetic datasets. This is because the real datasets are relatively small. Based on the experiment results, *ZSubspace* performs better than SUBSKY, when the number of skyline points is large.

8 Conclusion

We introduced Z-SKY that is the efficient skyline query processing framework in database systems based on our problem analysis and understandings about Z-order curve. Specifically, we analyzed the skyline query and pointed out its transitivity and incomparability properties that can facilitate search for skyline results. Correspondingly, we, in this paper, exploited the monotonic ordering and clustering features of the Z-order curve that perfectly match with these two properties. Because of the ordering feature, data points on the Z-order curve should have their dominating points if any accessed ahead of them, and thus the candidate reexamination is totally eliminated. Meanwhile, due to the clustering feature, data points with similar values are naturally mapped onto Z-order curve segments which are in turn grouped into blocks. It facilitates *true block-based dominance tests* that no other existing approaches can support and effective search space pruning. These two Z-order curve features are also explored for skyline result update and several skyline query variants. While results obtained from comprehensive experiments indicated the superiority of our approaches devised based on the Z-order curve features over all the existing works, we believe that these explored Z-order curve features and the proposed ZBtree index and extended ZBtree index are also useful for other skyline query variants not examined in this paper.

Acknowledgment This research was supported in part by the National Science Foundation under Grant no. IIS-0328881, IIS-0534343 and CNS-0626709.

References

1. Balke, W.-T., Güntzer, U., Zheng, J.X.: Efficient distributed skylining for web information systems. In: Proceedings of EDBT, pp. 256–273 (2004)
2. Bartolini, I., Ciaccia, P., Patella, M.: SaLSa: Computing the skyline without scanning the whole sky. In: Proceedings of CIKM, pp. 405–414 (2006)
3. Beyer, K.S., Goldstein, J., Ramakrishnan, R., Shaft, U.: When is "nearest neighbor" meaningful? In: Proceedings of ICDT, pp. 217–235 (1999)
4. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proceedings of ICDE, pp. 421–430 (2001)
5. 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 SIGMOD, pp. 503–514 (2006)
6. Chaudhuri, S., Dalvi, N.N., Kaushik, R.: Robust cardinality and cost estimation for skyline operator. In: Proceedings of ICDE, p. 64 (2006)
7. Chen, L., Lian, X.: Dynamic skyline queries in metric spaces. In: Proceedings of EDBT, pp. 333–343 (2008)
8. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with Presorting. In: Proceedings of ICDE, pp. 717–816 (2003)
9. Dellis, E., Vlachou, A., Vladimirskiy, I., Seeger, B., Theodoridis, Y.: Constrained subspace skyline computation. In: Proceedings of CIKM, pp. 415–424 (2006)
10. Fuhry, D., Jin, R., Zhang, D.: Efficient skyline computation in metric space. In: Proceedings of EDBT, pp. 1042–1051 (2009)
11. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. **30**(2), 170–231 (1998)
12. García, Y.J., Lopez, M.A., Leutenegger, S.T.: A greedy algorithm for bulk loading R-trees. In: Proceedings of ACM GIS, pp. 163–164 (1998)
13. Godfrey, P., Shipley, R., Gryz, J.: Maximal vector computation in large data sets. In: Proceedings of VLDB Conference, pp. 229–240 (2005)
14. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM TODS **24**(2), 265–318 (1999)
15. Huang, Z., Jensen, C.S., Lu, H., Ooi, B.C.: Skyline queries against mobile lightweight devices in MANETs. In: Proceedings of ICDE, p. 66 (2006)
16. Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: an online algorithm for skyline queries. In: Proceedongs of VLDB Conference, pp. 275–286 (2002)
17. Lee, K.C.K., Zheng, B., Li, H., Lee, W.-C.: Approaching the skyline in Z order. In: Proceedings of VLDB Conference, pp. 279–290 (2007)
18. Li, H., Tan, Q., Lee, W.-C.: Efficient progressive processing of skyline queries in peer-to-peer systems. In: Proceedings of Infoscale, p. 26 (2006)
19. Lin, X., Yuan, Y., Wang, W., Lu, H.: Stabbing the sky: efficient skyline computation over sliding windows. In: Proceedings of ICDE, pp. 502–513 (2005)
20. Lin, X., Yuan, Y., Zhang, Q., Zhang, Y.: Selecting stars: the k most representative skyline operator. In: Proceedings of ICDE, pp. 86–95 (2007)
21. Orenstein, J.A., Merrett, T.H.: A class of data structures for associative searching. In: Proceedings of PODS, pp. 181–190 (1984)
22. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. ACM TODS **30**(1), 41–82 (2005)
23. Pei, J., Jiang, B., Lin, X., Yuan, Y.: Probabilistic skylines on uncertain data. In: Proceedings of VLDB Conference, pp. 15–26 (2007)
24. Ramsak, F., Markl, V., Fenk, R., Zirkel, M., Elhardt, K., Bayer, R.: Integrating the UB-tree into a database system kernel. In: Proceedings of VLDB Conference, pp. 263–272 (2000)
25. Sharifzadeh, M., Shahabi, C.: The spatial skyline queries. In: Proceedings of VLDB Conference, pp. 751–762 (2006)
26. Tan, K.-L., Eng, P.-K., Ooi, B.C.: Efficient Progressive Skyline Computation. In: Proceedings of VLDB Conference, pp. 301–310 (2001)
27. Tao, Y., Ding, L., Lin, X., Pei, J.: Distance-based representative skyline. In: Proceedings of ICDE, pp. 892–903 (2009)
28. Tao, Y., Papadias, D.: Maintaining sliding window skylines on data streams. IEEE TKDE **18**(2), 377–391 (2006)
29. Tao, Y., Xiao, X., Pei, J.: Efficient skyline and top-k retrieval in subspaces. IEEE TKDE **19**(8), 1072–1088 (2007)
30. Wang, S., Ooi, B.C., Tung, A.K.H., Xu, L.: Efficient skyline query processing on peer-to-peer networks. In: Proceedings of ICDE, pp. 1126–1135 (2007)
31. Wu, P., Agrawal, D., Egeciglu, O., Abbadi, A.E.: DeltaSky: optimal maintenance of skyline deletions without exclusive dominance region generation. In: Proceedings of ICDE, pp. 486–495 (2007)
32. Wu, P., Zhang, C., Feng, Y., Zhao, B.Y., Agrawal, D., Abbadi, A.E.: Parallelizing skyline queries for scalable distribution. In: Proceedings of EDBT, pp. 112–130 (2006)
33. Yiu, M.L., Mamoulis, N.: Efficient processing of top-k dominating queries on multi-dimensional data. In: Proceedings of VLDB Conference, pp. 483–494 (2007)
34. Yuan, Y., Lin, X., Liu, Q., Wang, W., Yu, J.X., Zhang, Q.: Efficient computation of the skyline cube. In: Proceedings of VLDB Conference, pp. 241–252 (2005)
35. Zheng, B., Lee, K.C.K., Lee, W.-C.: Location-dependent skyline query. In: Proceedings of MDM, pp. 148–155 (2008)