Approaching the Skyline in Z Order

Ken C. K. Lee[†] cklee@cse.psu.edu
Baihua Zheng[‡] bhzheng@smu.edu.sg
Huajing Li[†] huali@cse.psu.edu
Wang-Chien Lee[†] wlee@cse.psu.edu

[†]Pennsylvania State University, University Park, PA16802, USA.
[‡]Singapore Management University, Singapore.

ABSTRACT

Given a set of multidimensional data points, skyline query retrieves a set of data points that are not dominated by any other points. This query is useful for multi-preference analysis and decision making. By analyzing the skyline query, we observe a close connection between Z-order curve and skyline processing strategies and propose to use a new index structure called *ZBtree*, to index and store data points based on Z-order curve. We develop a suite of novel and efficient skyline algorithms, which scale very well to data dimensionality and cardinality, including (1) *ZSearch*, which processes skyline queries and supports progressive result delivery; (2) *ZUpdate*, which facilitates incremental skyline result maintenance; and (3) *k-ZSearch*, which answers k-dominant skyline query (a skyline variant that retrieves a representative subset of skyline results). Extensive experiments have been conducted to evaluate our proposed algorithms and compare them against the best available algorithms designed for skyline search, skyline result update, and k-dominant skyline search, respectively. The result shows that our algorithms, developed coherently based on the same ideas and concepts, soundly outperforms the state-of-the-art skyline algorithms in their specialized domains.

1. INTRODUCTION

Given a set of d-dimensional data points, *skyline query* retrieves a set of data points that are *not dominated* by any other points. A point p is said to dominate another point p' if p is not worse than p' on all d dimensions and p is strictly better than p' on at least one dimension. Owing to a very large application base, skyline query has received a lot of attentions in the database community [3, 10, 13, 15]. It is currently promoted for incorporating into commercial database systems [6] as well.

Most of the work in the literature targets at improving the performance of finding *skyline points* from a very large dataset [3, 10, 13, 15, 17]. Among all the performance criteria, *search efficiency* and *update efficiency* are two most important ones. Additionally, variants of skyline queries have been proposed, e.g., *k-dominant skyline* query [5] that retrieves a representative subset of skyline points from a high-dimensional data set. It relaxes the dominant condition by considering k among d dimensions. While various solutions have been proposed to address those different but closely related research issues, almost all of them are developed independently. This work develops a generic yet highly efficient solution in support of skyline queries as well as skyline updates, and demonstrates the flexibility of our solution to answer *k-dominant skyline* query coherently[1]. To the best of our knowledge, this is the first work to address both the efficiency and the flexibility aspects in skyline solutions in high data dimensional spaces.

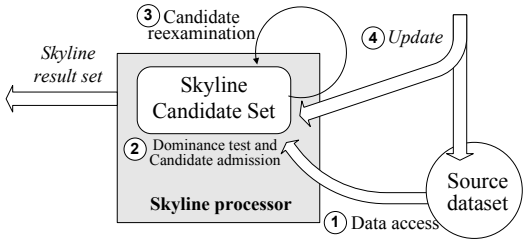


Figure 1: General skyline processing framework

Figure 1 shows a general framework commonly adopted by existing works for skyline query processing, e.g., [7, 13, 15]. Under this framework, a skyline processor needs to carry out three tasks to locate skyline points from a source dataset: **(1)** Data access: loading potential skyline data points from the source dataset for *dominance tests*[2]; **(2)** Candidate admission: admitting a skyline candidate to the Candidate Set if it passes the dominance test, otherwise discarding it; and **(3)** Candidate reexamination: performing dominance tests on existing candidates against new candidates to filter out false candidates. After the processing, all the remaining skyline candidates are the final skyline points.

In the above framework, an update (i.e., an insertion or a deletion) performed at the source dataset is also submitted to the skyline processor (shown as **(4)** in Figure 1). An inserted data point can be processed by first performing a dominance test (i.e., **(2)**) and then if admitted, eliminating existing skyline points dominated by the newly inserted skyline (i.e., **(3)**). However, when a skyline point is deleted, some data points in the source dataset may become non-dominated and thus are *promoted* to the skyline candidate/result set. In this case, all the tasks **(1)(2)(3)** need to be invoked to locate qualified data points for promotion. Additionally, this framework can be adjusted to support k-dominant skyline by conducting tasks **(2)** and **(3)** based on the k-dominance condition instead of conventional dominance condition.

Skyline processing is obviously an expensive operation. Both the I/O cost for accessing data from external storage and the CPU cost for dominance test contribute significantly to the total processing cost. Even though the processing time is typically dominated by the I/O cost in database systems, the CPU time spent to perform pairwise dominance tests here in skyline processing is not negligible. Through our analysis, we have obtained several observations:

[1]We plan to support other skyline variants in the future.
[2]A dominance test compares data points on a dominance condition.

VLDB '07, September 23-28, 2007, Vienna, Austria.

1. The access order of data points has a direct impact on the performance. Identifying skyline points that dominate many data points early can eliminate unqualified data points from further examination. Moreover, certain access order of data points may avoid candidate reexaminations.
2. Pairwise point-to-point dominance test is very time-consuming. Efficient query processing and result update algorithms should accelerate the dominance tests and facilitate search space pruning in terms of *blocks* of data points.
3. The organization of skyline candidates and their examination strategies are critical for the efficiency of dominance test.

Following the design principles derived from the above observations, we adopt an approach based on *Z-order curve* that carries many good properties for processing skyline queries. Z-order curve orders the data points and clusters them in blocks to facilitate efficient dominance tests and space pruning. A new index, namely, *ZBtree*, is proposed to store data points and maintain the skyline candidate set. A suite of algorithms, namely, *ZSearch*, *ZUpdate* and *k-ZSearch*, are developed to process skyline queries, skyline result updates, and k-dominant skyline queries, respectively. Finally, we conducted a comprehensive performance evaluation to compare our proposal with existing algorithms.

The remainder of this paper is organized as follows. In Section 2, we review the skyline problem and existing works. In Section 3, we analyze the issues in skyline processing and explain why adopting Z-order curve is an elegant solution, which leads to the introduction of ZBtree to serve as the index of our algorithms. Section 4, 5 and 6 detail ZSearch, ZUpdate and k-ZSearch algorithms, respectively. Section 7 evaluates the performance of our algorithms. Finally, Section 8 concludes the paper.

2. PRELIMINARIES

This section first discusses the skyline problems and associated properties, and then reviews closely related works.

2.1 Skyline Problems and Properties

Given a d-dimensional space $S=\{s_1, s_2, \cdots s_d\}$ and a set of data points $P=\{p_1, p_2, \cdots p_n\}$ (where p_i is a data point on S), the dominance conditions and skyline problems are defined. We use $p_i.s_j$ to denote the j-th dimensional value of p_i. We assume the existence of a total order relationship, either '$<$' or '$>$', on each dimension. Without loss of generality, we consider '$<$' relationship in this paper.

DEFINITION 1. **Dominance**. *Given $p, p' \in P$, p dominates p' iff* $\forall s_i \in S, p.s_i \leq p'.s_i \wedge \exists s_j \in S, p.s_j < p'.s_j$. □

We denote that p dominates p' by $p \vdash p'$ and use $p \nvdash p'$ to represent that p does not dominate p'.

DEFINITION 2. **Skyline**. *A data point $p \in P$ is a skyline point in S iff p is not dominated by any other point $p'(\neq p) \in P$, i.e., $\nexists p' \in P - \{p\}, p' \vdash p$.* □

We observe two important properties of the skyline problem, namely, *transitivity* and *incomparability*, which provide important insights to facilitate our algorithm development.

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., $p \nvdash p' \wedge p' \nvdash p$.* □

Based on the transitivity, if dominating points are always processed before their dominated data points, a data point which passes the dominance tests against all the skyline points ahead of it is guaranteed to be a skyline point. This property inspires the sorting-based approaches [7, 8]. If two data points are known to be incomparable, dominance tests between them can be avoided. This incomparability inspires division-based approaches [3]. In addition, it is also applicable to dominance tests on blocks of data points (and thus can reduce the expensive individual point-to-point comparisons).

To retrieve a representative subset of skyline points, k-dominant skyline query has been proposed [5]. With k-dominant skyline query, data points with few good attributes would be dominated and excluded from the result, based on the following k-dominance condition. We denote that p k-dominates p' by $p \vdash_k p'$ and that p does not k-dominate p' by $p \nvdash_k p'$.

DEFINITION 3. **k-dominance**. *Given $p, p' \in P$, p k-dominates p' iff* $\exists S' \subseteq S, |S'| = k, \forall s_i \in S', p.s_i \leq p'.s_i \wedge \exists s_j \in S', p.s_j < p'.s_j$. □

DEFINITION 4. **k-dominant skyline**. *A data point p is a k-dominant skyline point iff there is no point p' ($\neq p$) that k-dominates p, i.e., $\nexists p' \in P - \{p\}, p' \vdash_k p$.* □

For k-dominate skyline query, it is important to note that the transitive property (Property 1) is no longer valid. 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. In this case, both p and p' are not k-dominant skyline points, which is the so-called *cyclic dominance* [5].

2.2 Related Work

In the following, we review representative algorithms for skyline query, skyline result update and k-dominant skyline query. While these algorithms are designed for specific problems, our suite of algorithms, developed in the same design principles, can tackle all of the addressed problems efficiently. In addition to the closely related work to our research on centralized databases, many other studies extend skyline queries in various environments, e.g., data stream [11, 16], distributed databases [18], Web [1], Mobile Ad hoc Network (MANET) [9], etc.

2.2.1 Skyline Query Processing

Skyline query processing algorithms, including Block-Nested Loop (BNL) [3], Bitmap [15], Divide-and-Conquer (D&C) [3], Sort-Filter-Skyline (SFS) [7], LESS [8], Index [15], Nearest Neighbor (NN) [10], and Branch and Bound Search (BBS) [13], can be roughly classified by the techniques used, i.e., *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.

	BNL	Bitmap	D&C	SFS,LESS	Index	NN	BBS
D&C	χ	χ	✓	χ	✓	✓	✓
Sort	χ	χ	χ	✓	✓	✓	✓

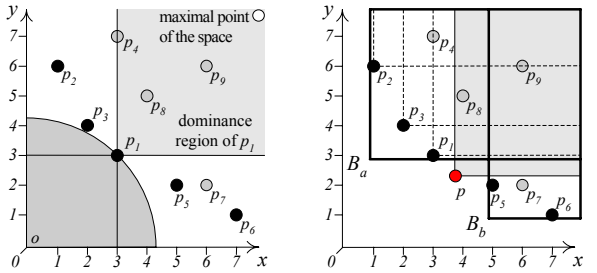
Table 1: Classification of skyline query algorithms

Divide and Conquer Algorithms. D&C divides a dataset into several partitions small enough to be loaded into the main memory for processing [3]. A partial skyline for each partition is computed. Then, the complete skyline is obtained by merging all partial skylines and removing dominated data points.

Sorting-based Algorithms. SFS is devised based on an observation that by getting a dataset presorted according to a certain monotone scoring function such as entropy, or sum of attributes, data points are guaranteed not to 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. Dominance tests in SFS are based on an *exhaustive scan* on existing skyline candidates. Unless the number of

skyline points is very small, SFS tends to be CPU-bound. LESS 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 if the number of skyline points is small. These sorting-based approaches have no search space pruning capability and inevitably examine all the data points.

Hybrid Algorithms. Hybrid algorithms, including Index, NN and BBS, use both divide-and-conquer and sorting techniques in skyline processing. We review BBS, the existing most efficient skyline search algorithm below. Generally speaking, BBS is based on iterative NN search. Figure 2(a) illustrates BBS with 9 2D points. In the figure, p_1, the nearest neighbor (nn) to the origin in the whole space, is located as the first skyline point. 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 and can be discarded safely. The second nn, p_3, not dominated by p_1, is another skyline point. Similarly, 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.



(a) BBS (b) Main memory R-tree

Figure 2: BBS and Main memory R-tree

BBS adopts R-tree as its underlying index structure for its efficient nn search. It also uses a *heap* that keeps track of unexamined data points and index nodes in distance order to alleviate repeated access of upper portion of the R-tree and the *main memory R-tree* to index the dominance regions of all skyline points. Via a main memory R-tree, BBS performs dominance tests on an examinee data point (or index node) by issuing an enclosure query. If the examinee is entirely enclosed by any skyline candidate's dominance region, it is dominated. In Figure 2(b), p_8, an examinee data point, is compared with B_a and B_b, intermediate index nodes' MBBs in the main memory R-tree. As p_8 is enclosed in B_a, not B_b, p_8 may be dominated by those in B_a but not B_b. Finally, p_8 is compared with all data points inside B_a and found dominated by p_1 (or p_3).

However, the performance of R-tree that BBS depends on deteriorates when data dimensionality increases (due to the curse of dimensionality [2]). Even worse, some data points and index nodes are prematurely loaded into the heap, overconsuming the runtime memory. Our proposed ZSearch (see Section 4) is also a hybrid approach. It processes the skyline by accessing data points in Z-order, a widely received dimension reduction technique [14], where data points are organized in a sorted and clustered fashion, improving the processing time and the runtime memory consumption. With block-based dominance tests, ZSearch can efficiently assert if a block of data points is dominated.

2.2.2 Skyline Result Updates

Compared with skyline query processing, skyline result update is a relatively new research area. In general, updates include insertions and deletions. Insertions may bring in new skyline points. In BBS-Update [13][3], a new data point is compared with dominance regions of skyline candidates indexed by a main memory R-tree. It first determines if the data point is dominated by any existing skyline point. If a new data point is not dominated, it is enrolled to the main memory R-tree and those existing skyline points dominated by this new point are removed. However, main memory R-tree is inefficient to identify dominated skyline points. For example, in Figure 2(b), 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 enclosed data points. Our ZUpdate utilizes the sorting property of Z-order curve to quickly figure out portions of skyline points not dominated by the newly inserted point and portions need to perform dominance tests.

Deletion is more complicate than insertion since data points previously dominated by a deleted point may no longer be dominated and thus get promoted to the 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 associated skyline point. However, in high dimensional spaces, EDR could be in irregular and complex shape. To efficiently determine whether a data point p (or an index node) is exclusively dominated by the deleted point, DeltaSky [17] extends BBS-Update by maintaining d sorted lists, each corresponding to one dimension. Each list sorts the data points according to their values on a particular dimension. If p is dominated, its dominating skyline point(s) should be smaller than or equal to p in all d sorted lists. Based on this idea, DeltaSky performs a negative test. It scans all the lists and determines if all skyline points can be found greater than p among all the lists. However, DeltaSky also suffers from a serious scalability problem to high data dimensionality. First, it needs to scan d lists in a high dimensional space. The number of skyline points is expected to be large [5], which extends the length of all sorted lists. Second, DeltaSky does not address insertion. Sorted lists favor deletions but they incur update overheads to insertions. Our ZUpdate (shown in Section 5) can efficiently handle both insertion and deletion (and even multiple deletions simultaneously).

2.2.3 k-Dominant Skyline Query Processing

By relaxing the dominance condition to consider k among d dimensions, k-dominant skyline query [5] reduces the size of skyline result set. However, data points can k-dominate each other simultaneously. Figure 3 lists 4 3D 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 2-dominant skyline, p_1 and p_2 2-dominate each other. This cyclic dominance relationship violates the transitivity property (Property 1), making all existing skyline search algorithms inapplicable.

	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

Figure 3: k-dominant skyline example

In [5], three algorithms for k-dominant skyline query are proposed, namely, One-Scan-Algorithm (OSA), Two-Scan-Algorithm (TSA) and Sort-Retrieval-Algorithm (SRA). OSA scans the dataset once. It maintains both k-dominant skyline points and conventional skyline points to handle cyclic dominance. However, exhaustive comparisons substantially slow down OSA. TSA improves OSA by maintaining only k-dominant skyline but it scans the dataset twice. The first scan collects candidate points, which might include false hits, and the second scan eliminates those false hits. In Figure 3, 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. p_4, not 2-dominated after the second scan, is the 2-dominant skyline. RSA is an elimination based approach. It initially puts all data points as k-dominant skyline point candidates. In addition, all

[3]We use BBS-Update to refer to the update algorithm and BBS for search algorithm.

of them are stored in d sorted lists. Iteratively, it loads one data point from one sorted list and compares it with all the candidates. Those dominated candidates are removed. Finally, the remaining candidates are k-dominant skyline points. Among all approaches, TSA outperforms OSA and SRA as reported in [5].

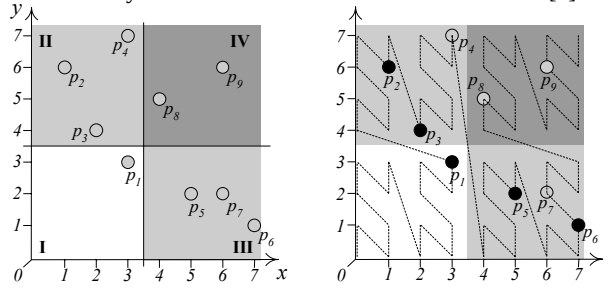
With the relaxed k-dominant condition, more data points can be dominated. The search space thus is significantly shrunk. However, those three existing algorithms, not conscious of this fact, have to access all individual data points. Obviously, points p_1, p_2 and p_3 in Figure 3 can be grouped as a block (i.e., $\langle\langle 2, 9\rangle, \langle 8, 11\rangle, \langle 2, 11\rangle\rangle$) 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 (shown in Section 6) approaches this k-dominant skyline search problem by exploiting the clustering property of Z-order curve. When a cluster of data points is found to be k-dominated, the examination of the enclosed points is avoided, improving the search performance.

3. SKYLINE PROCESSING IN Z-ORDER

In this section, we first summarize some unique properties of the Z-order curve that are observed to match perfectly well with some processing strategies for skyline query. We develop ZBtree, an index structure based on Z-order curve to facilitate searches/updates.

3.1 Skyline and Z-Order Curve

The efficiency of skyline processing and result updating is highly dependent on the three observations discussed in Section 1, two of which are the invocation of *dominance tests* and the *access order of the data points*. It is desirable to adopt block-based dominance tests that quickly eliminate unnecessary pairwise point comparisons. The access order is important because the early identification of skyline points that dominate lots of other data points can eliminate many candidate examination and reexamination [7].



(a) 2D data space (b) Z-order curve

Figure 4: Z-order curve example

Z-order space filling curve [14] has very nice geometric properties that fit the skyline processing strategies. For illustration, Figure 4(a) depicts a running example with 9 2D data points ($p_1, \cdots p_9$). Suppose we partition the entire space into four equal-sized quadrants, i.e., regions I, II, III and IV, along directions parallel to the two axis. Data points in Region I are not dominated by data points in the other three regions. On the contrary, all data points in Region IV are dominated by any point in Region I. In other words, as long as Region I is non-empty, all the points located inside Region IV can be discarded from examination. In addition, Region II and III are diagonally opposite to each other, and their data points do not dominate each other (i.e., data points in Region II and those in Region III are not comparable). Dominance tests between them can be avoided. However, points in Regions II or III may get dominated by some data points in Region I. These observations provide excellent heuristics for dominance tests in regional (i.e., block) level.

These observations also lead to a natural access order of the regions (and their data points) for processing skyline search, 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 employed to subregions divided from each region. The finest subregion can be small enough to cover only one distinct coordinate. This access sequence exactly follows a (rotated) 'Z' order, which is the well known Z-order space filling curve. Figure 4(b) shows the Z-order curve, which starts at the origin and passes through all coordinates (and data points) in the space once. This curve can also be easily extended to a *high-dimensional* space.

Like other space filling curves, Z-order curve maps data points in a multidimensional space to a one-dimensional space, with each point represented by a unique number, 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 domain ranges, the Z-address of a data point contains dv bits, which can be considered as v d-bit groups. The i-th 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., (001,110) in binary) is 010110. Here, 01, 01, 10 are the three 2-bit groups. 01 is formed by the first bit of x-coordinate 1 (i.e, 0 of 001) and that of y-coordinate 6 (i.e., 1 of 110). Similarly, the Z-address of p_4 (3,7) (i.e.,(011,111) in binary) is 011111 and 01, 11, 11 are the three 2-bit groups. This Z-address calculation is reversible so the original coordinate can be recovered from its corresponding Z-address[6].

Z-addresses in nature are hierarchical. 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 subspace into 2^d equal-sized smaller sub-spaces, and so on. For instance, p_2 and p_4 have 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 left-upper quad of the 2-D space). Based on Z-addresses, the Z-order curve provides two important properties, *monotonic ordering* and *clustering*, as stated below. They perfectly match transitivity and incomparability properties of skyline problem already discussed in Section 2.1.

PROPERTY 3. **Monotonic Ordering**. *Data points ordered by non-descending Z-addresses are monotonic in a way that a dominating point is placed before its dominated points.* □

As shown in Figure 4(b), based on Z-order curve, p_1 that dominates p_8 and p_9 is accessed before both of them. 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.

PROPERTY 4. **Clustering**. *Data points ordered by Z-addresses are naturally clustered as regions.* □

Due to the hierarchical property of Z-addresses, data points located in the same regions have similar Z-addresses. For instance, p_2, p_3 and p_4 (with the first bit-group (i.e., 01) in common) are located closely in the same region. Similarly, p_5, p_6, and p_7 having the same first bit group (i.e. 10) are located in another region. Grouping data points in regions can facilitate block-based dominance tests. When two regions are incomparable, dominance tests between points from each region can be safely eliminated. Moreover, once a point inside the region that dominates another region is identified, all the points within the dominated region can be safely omitted from dominance tests. For instance, p_8 and p_9 inside region IV can be safely omitted once p_1 that dominates them is identified.

Note that other space filling curves, e.g., Hilbert and Peano curves, are not suitable for skyline processing due to the lack of monotonic ordering property. These curves do not always start at the origin of a space (subspace), i.e., dominating points may be placed after their dominated points, and thus require reexamination.

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

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

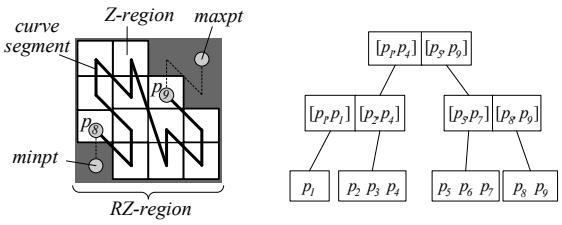
[6]We may use Z-address and coordinate interchangeably when the context is clear.

3.2 ZBtree Index Structure

As indicated above, Z-order curve provides two important properties, i.e., monotonic ordering and clustering, for skyline processing. Thus, to facilitate efficient processing of skyline query, we aim at designing an index structure that maintains these properties. As we map multidimensional data points onto one-dimensional addresses, a natural indexing approach is to combine Z-order curve and B$^+$-tree, e.g., based on the seminal work of Orenstein and Merett [12] or the more recent UB-tree [14]. However, those works mostly focus on range queries that find data points within a specified rectangular region. On the other hand, focusing on finding skyline points, our goals for such an index is to i) facilitate data processing in Z-order sequence; and ii) preserve data points in blocks to enable efficient search space pruning. Meanwhile, we aim at further exploiting the Z-order properties and providing theoretical foundation for advanced skyline processing. The indexes developed in [12, 14], while based on Z-order curve and B^+-tree, do not achieve our goals, e.g., [12] keeps regions as indexing information rather than a bounding region as to enables efficient search space pruning (to be shown later). Thus, we propose ZBtree, a new variant of B$^+$-tree, to organize data points in accordance with monotonic Z-addresses. ZBtree strategically divide a Z-order curve into disjoint segments (i.e., sequences of data points on the curve), with each of which represents a region. As such, the clustering property is preserved.

In ZBtree, leaf nodes maintain data points (i.e., Z-addresses) and non-leaf nodes maintain intervals (denoted by $[\alpha, \beta]$) which represent curve segments bounding the enclosed data points. The space covered by a Z-order curve segment is called *Z-region*. For example, the region passed by the curve staring at point p_8 and ending at point p_9 is a Z-region, as shown in Figure 5(a). 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 5.

DEFINITION 5. *An RZ-region is the smallest square area covering a Z-region bounded by $[\alpha, \beta]$. RZ-region is specified by two Z-addresses, $minpt$ and $maxpt$, so $[\alpha, \beta] \subseteq [minpt, maxpt]$.* □



(a) RZ-region (b) ZBtree

Figure 5: Example RZ-region and ZBtree

RZ-Regions can be easily derived based on Z-Regions (i.e., $[\alpha, \beta]$). First, the common prefix (i.e., some beginning d-bit groups) of both α and β is retrieved. With the common prefix, $minpt$ is formed by appending all 0's to the rest of bits and $maxpt$ is formed by setting all remaining bits to 1's. Figure 5(a) shows an RZ-region derived from the Z-region bounded by points p_8 and p_9. While forming RZ-regions is straightforward, it has the following property that can considerably boost the skyline search performance.

PROPERTY 5. *Within an RZ-region, all data points (except the one at $minpt$) are dominated by the data point at $minpt$; and the data point located at $maxpt$ is dominated by any point (except the one at $maxpt$).* □

As we shall discuss later, a ZBtree node can be safely discarded from dominance tests if its corresponding RZ-region is found to be dominated, thereby reducing comparison overheads. The search efficiency depends on the way data points are organized to form RZ-regions. To minimize the storage overhead of ZBtree, all data points can be tightly packed. However, this strategy would result in large RZ-regions which do not help much in pruning search space. Following our example in Figure 4(b), if a leaf node contains at most 3 data points, $p_1, \cdots p_9$ are allocated into 3 separate leaf nodes. Among them, p_7, p_8 and p_9 are put in one node and the corresponding RZ-region turns out to cover the entire space as those three points do not share any common prefix. Since this big RZ-region will not be dominated by any data point, the corresponding leaf node together with all the enclosed data points need to be accessed. As we observe from Figure 4(b), points p_8 and p_9 can be discarded once we identify the point p_1. Therefore, we strategically trade some storage overhead for processing efficiency by putting as many data points belonging to the same RZ-region as possible into a node (instead of filling up the entire node capacity). Assume that the minimum leaf node capacity is 1. 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 Figure 5(b). Although this requires some storage overhead, the search performance is significantly improved (as unnecessary node traversal and comparisons between incomparable nodes are avoid). Based on this principle, we group leaf nodes into appropriate non-leaf nodes and recursively propagate the process upwards till the root is reached.

3.3 ZBtree Index Manipulation

The primary objectives of ZBtree manipulations, including insertion, deletion and bulkloading, are to maintain the monotonic and clustering properties of ZBtree, while keeping the size of ZBtree reasonably small. In the following, we detail the insertion, deletion and bulkloading operations. We assume each node contains at most N entries and the minimum node utilization threshold is M (where $M \leq N/2$).

Insertion places a data point with Z-address z_{ins} to a leaf node. The search for the target leaf node is based on depth-first traversal 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 disjoint Z-order curve segments such that the total area of their corresponding RZ-regions is 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 from the leaf node to its parent/ascendent nodes.

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 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 ZBtree. Similarly, the intervals of the nodes along the deleted leaf node to the root node are updated to reflect the deletion.

Bulkloading builds a ZBtree in a bottom-up fashion. It has two steps. The first step sorts all the data points based on the non-decreasing Z-addresses. The second step scans these data points sequentially using a sliding window with N slots to form nodes. Within the window, the largest number of data points (denoted as x) to form a small RZ-region are put into a leaf node, where $M \leq x \leq N$. Specifically, we put all the sorted data points into a stack and pop out the top N data points into the sliding window. First, we form a RZ-region R using those N data points in the window. Then, we, in each trial, push back the point with the largest Z-address (i.e., decrement x by 1 where x is initialized to N) to the

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

stack and determine the corresponding RZ-region R'_x. If R'_x is smaller than R, those x data points in the sliding window form a node. If no R'_x (for all tried x) is smaller than R, we pop out all $N - x$ data points to form a node. As such, high space utilization is guaranteed and formation of small RZ region is favored. This process repeats until all the data points are examined. Then all leaf nodes are examined in the same fashion to generate non-leaf nodes. This process continues until a single node, the root, is formed.

4. ZSEARCH FOR SKYLINE QUERY

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

4.1 RZ-Region Based Dominance Test

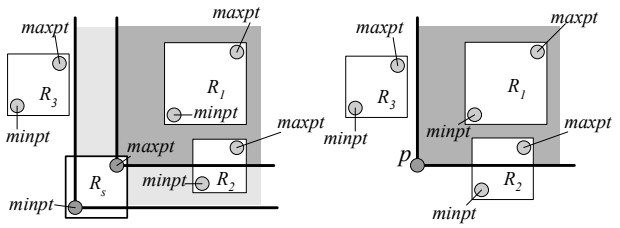
Dominance test is a key determinant of computational overhead. Comparing data points pairwise is very time consuming. To alleviate this cost, we introduce block based dominance test in ZSearch based on RZ-regions derived during traversal in ZBtree. In ZSearch, we maintain one ZBtree for storing source data set (labelled as $\mathcal{SRC}$) and one ZBtree for maintaining skyline points (labelled as $\mathcal{SL}$). Both ZBtrees provide RZ-regions. The dominance condition comparing two RZ-regions is defined in Lemma 1. When not causing any confusion, we use RZ-region R to denote all the data points within it and use $minpt(R)$ and $maxpt(R)$ to refer to $minpt$ and $maxpt$ of R, respectively.

LEMMA 1. *Given two RZ-regions R and R', the following three cases hold:*

1. *If $maxpt(R) \vdash minpt(R')$, then $R \vdash R'$.*
2. *If $maxpt(R) \nvdash minpt(R') \wedge minpt(R) \vdash maxpt(R')$, then some points in R' may be dominated by others in R.*
3. *If $minpt(R) \nvdash maxpt(R')$, then $R \nvdash R'$.* □

Proof. We prove the lemma case by case.
Case 1. $maxpt(R) \vdash minpt(R')$**.** By transitivity (Property 1), $\forall p \in R - \{maxpt(R)\}, \forall p' \in R' - \{minpt(R')\}, p \vdash maxpt(R) \wedge minpt(R') \vdash p' \Rightarrow p \vdash p'$ and as stated, $maxpt(R) \vdash minpt(R')$. Hence, $R \vdash R'$. R_s and R_1 in Figure 6(a) form an example.
Case 2. $maxpt(R) \nvdash minpt(R') \wedge minpt(R) \vdash maxpt(R')$**.** Some data points (e.g. $maxpt(R')$) are dominated by $minpt(R)$. Thus, the case holds. R_s and R_2 in Figure 6(a) form an example.
Case 3. $minpt(R) \nvdash maxpt(R')$**.** This case can be proved by contradiction. Assume $p \vdash p'$ ($p \in R$, $p' \in R'$). 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_3 in Figure 6(a) form an example. ■

This dominance condition is generic enough to be applied to an RZ-region which contains a single data point p, i.e., $minpt(R) = maxpt(R) = p$. So, the dominance condition between two points defined in Definition 1 in Section 2.1 is a special case of this lemma. Figure 6(b) shows the dominance tests of R_1, R_2, R_3 against p.



(a) Two RZ-regions (b) RZ-region against a point

Figure 6: Dominance test

Based on Lemma 1, we can perform effective dominance tests on the RZ-region of any node from $\mathcal{SRC}$ against that from $\mathcal{SL}$. Figure 7 shows Algorithm $Dominate$ 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 (i.e., those available in $\mathcal{SL}$). The algorithm traverses $\mathcal{SL}$ based on *breadth-first traversal* such that RZ-regions of high-level nodes from $\mathcal{SL}$ are compared against R and drilled down if R needs further examination against finer RZ-regions in $\mathcal{SL}$.

Algorithm $Dominate(\mathcal{SL}, minpt, maxpt)$
Input. $\mathcal{SL}$: 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** children 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 $mint \vdash maxpt$) **then**
10. q.enqueue(c); /* *Case 2 of Lemma 1* */
11. **else** /* *leaf node* */
12. **forall** children point c of n **do**
13. **if** ($c \vdash minpt$) **then**
14. **output** TRUE;
15. **output** FALSE;
End.

Figure 7: Algorithm $Dominate$

In the algorithm, 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. If it may contain some points dominating R (i.e., Case 1 of Lemma 1), the algorithm indicates R is dominated and the dominance test is completed (Line 7-8). If the RZ-region of n may dominate some but not all points inside R, (i.e., Case 2 of Lemma 1), the entry is queued for further examination (Line 9-10). Case 3 of Lemma 1 is omitted since it is implied by the failures of the above two cases and there is no further comparison needed as these two RZ-regions are found incomparable.
- n is a leaf node. We check R against individual skyline points inside (Line 13-14).

This traversal continues until the queue is vacated (i.e., all relevant nodes in $\mathcal{SL}$ are visited). Finally, R is reported as not dominated. Note that the search might stop before the leaf level is reached. That means fewer nodes in $\mathcal{SL}$ need to be accessed and the efficiency of dominance test is enhanced.

4.2 ZSearch Algorithm

With the dataset organized in a ZBtree (i.e., $\mathcal{SRC}$), ZSearch traverses the tree to visit RZ-regions and potential skyline points in a depth-first order, which exactly follows the Z-order, with a stack keeping track of unexplored paths. The stack memory consumption is bounded by a factor of the tree height of the $\mathcal{SRC}$. Figure 8 depicts the pseudo-code of ZSearch. It fetches the index nodes/data points from $\mathcal{SRC}$ (Line 2-14). At each round, the RZ-region of a node is examined against $\mathcal{SL}$ by invoking Algorithm $Dominate$ (see Figure 7). If its corresponding RZ-region is not dominated, the node is further explored (Line 7-13). Data points not dominated by any skyline candidate in $\mathcal{SL}$ are collected and inserted to $\mathcal{SL}$ (Line 13). The organization of skyline candidates in $\mathcal{SL}$ enables formation of RZ-regions to facilitate dominance tests. $\mathcal{SL}$ may be too large to store in the main memory. It can be stored on disk and we use available memory as a cache. This approach is appropriate since 1) the cached upper (and usually a small) portion of $\mathcal{SL}$ can be sufficient to perform dominance tests without reaching the leaf

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

levels, and 2) clustered data points in $\mathcal{SRC}$ exhibit good access locality of related RZ-regions in $\mathcal{SL}$. The algorithm terminates when all entries from $\mathcal{SRC}$ are examined signaled by an empty stack.

Algorithm $ZSearch(\mathcal{SRC})$
Input. $\mathcal{SRC}$: ZBtree for source data set;
Local. s: Stack;
Output. $\mathcal{SL}$: ZBtree for skyline points;
Begin
1. s.push($source$'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** children node c of n **do**
9. s.push(c);
10. **else** /* *leaf node* */
11. **forall** children point c of n **do**
12. **if** not $Dominate(\mathcal{SL}, c, c)$ **then**
13. $\mathcal{SL}$.insert(c);
14. **output** $\mathcal{SL}$;
End.

Figure 8: Algorithm $ZSearch$

We use the data points in Figure 4(a) to illustrate the algorithm, with Figure 5(b) showing the corresponding ZBtree. The trace is depicted in Figure 9. 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 simplicity, only leaf nodes enclosed by $\{\}$ are shown. First, we obtain p_1, the first skyline point. Next, p_2, the second data point on the Z-order curve, incomparable to p_1, is inserted to $\mathcal{SL}$. Note that BBS accesses p_3 (not p_2) after p_1. Then, p_3 (incomparable to both p_1 and p_2) is inserted to $\mathcal{SL}$. Assume the node capacity of $\mathcal{SL}$ to be 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 that of $\{p_2, p_3\}$, the comparison with p_2 or p_3 is saved. Next, explored p_5 and p_6 are inserted to $\mathcal{SL}$ while p_7 is dropped. Finally, $[p_8, p_9]$ dominated by $\{p_1\}$ in $\mathcal{SL}$ is skipped and the search completes.

Stack	Skyline points $\mathcal{SL}$
$[p_1, p_4]$,$[p_5, p_9]$	$\emptyset$
$[p_1, p_1]$,$[p_2, p_4]$,$[p_5, p_9]$	$\emptyset$
$[p_2, p_4]$,$[p_5, p_9]$	$\{p_1\}$
$[p_5, p_9]$	$\{p_1\}$,$\{p_2, p_3\}$
$[p_5, p_7]$,$[p_8, p_9]$	$\{p_1\}$,$\{p_2, p_3\}$
$[p_8, p_9]$	$\{p_1\}$,$\{p_2, p_3\}$,$\{p_5, p_6\}$

Figure 9: $ZSearch$ **Trace**

4.3 Discussion

Here we discuss the performance of ZSearch in terms of the runtime memory consumption, the expected processing time and the I/O costs. Let n, d, m be the data cardinality, data dimensionality and the number of skyline points, respectively, where generally speaking $d \ll n$ and $m \leq n$. We assume $\mathcal{SRC}$ and $\mathcal{SL}$ share the same node capacity, which is a constant.

We first study the runtime memory consumption. Two data structures are used in ZSearch, i.e., a stack keeping track of unexplored paths and $\mathcal{SL}$ maintaining skyline result candidates. The maximum space required for stack is bounded by the height of $\mathcal{SRC}$ times node capacity, i.e., $O(log_d(n))$. In contrast, BBS uses a heap to maintain a number of unexplored nodes that is $O(m)$. Consequently, ZSearch is more space efficient than BBS in most cases. Meanwhile, the maximum size of $\mathcal{SL}$ depends on the number of candidates, i.e., $O(m)$, the same size as BBS's main memory R-tree. Note that dominance test may terminate at some upper-level nodes of $\mathcal{SL}$ but BBS has to reach the leaf nodes.

Next, we examine the processing time that is governed by several factors, the number of times algorithm $Dominate$ invoked, the overhead of algorithm $Dominate$, and the I/O costs in terms of number of disk pages accessed. Those vary with respect to data distributions, namely, correlated, anti-corrected and independent [3]. Based on these, we analyze the performance.

Correlated data distribution. In this case, only a few skyline points are resulted while the majority of data points are dominated. Consider a case that there is only one skyline point, i.e., $m = 1$. The skyline point should be the first accessed data point due to monotonic ordering property of Z-order curve. Before retrieving this data point, $\mathcal{SL}$ is empty, so the overhead of algorithm $Dominate$ is zero. Then the first data point (i.e., skyline point) is inserted to $\mathcal{SL}$. All other data points and index nodes in the stack are examined by $Dominate$ and get dominated. The overhead of $Dominate$ is 1 while examinee entries (data points and index nodes) are on the path from the root to the first leaf node, which is related to the height of the tree, i.e., $O(log_d(n))$. Since each dominance comparison considers d dimensions, the processing time is $O(d \cdot log_d(n))$. Meanwhile, the I/O cost is equivalent to the height of the tree, that is $O(log_d(n))$.

Anticorrelated data distribution. Here, most of data points are not dominated. Our analysis considers the worst case that all data points are not dominated and they all are skyline points, i.e., $m = n$. The number of times algorithm $Dominate$ invoked equals to the total number of nodes and data points, i.e., $O(n)$. On the other hand, the overhead of each $Dominate$ call is $O(log_d(n))$, since the traversal in $\mathcal{SL}$ does not necessarily visit all branches and it could terminate at a high-level node of $\mathcal{SL}$. As a result, the processing time is $O(n \cdot d \cdot log_d(n))$ and I/O cost is $O(d \cdot n)$.

Independent data distribution. For this data distribution, m out of n data points are skyline points. As suggested in [4], $m = \Theta((log(n))^{d-1}/(d-1)!)$ that is highly dependent on the data dimensionality, d. Based on this, given $n = 1000$k, if $d = 2$, m is 6k (0.6%); if $d = 9$, m is 210k (21%); and if $d \geq 10$, $m = n$. For high dimensionality, the processing time and the I/O cost become close to $O(n \cdot d \cdot log_d(n))$ and $O(n)$, respectively. For low dimensionality, a few data points are skyline points and only the branches from the root to those leaf nodes containing them are examined. Then, the processing time and I/O cost of ZSearch are respectively $O(d \cdot log_d(n) \cdot log_d(m))$ and $O(log_d(n))$, as only some paths from the root to skyline result points are examined while many branches are found dominated at the upper level of tree and hence are pruned.

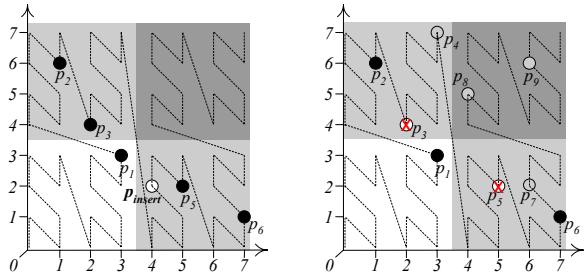
5. ZUPDATE FOR SKYLINE UPDATES

It is clearly inefficient to reevaluate a skyline query every time when the underlying dataset is changed. This section presents ZUpdate algorithm, which incrementally updates the skyline results using two previously discussed ZBtrees (i.e., $\mathcal{SRC}$ and $\mathcal{SL}$). $\mathcal{SL}$ is generated during ZSearch. An update on a dataset can be an insertion of a new data point or a deletion of an existing one. As modification of a data point can be regarded as deletion of the data point (i.e. old value) followed by insertion of another point (i.e. new value), we only discuss insertion and deletion below.

5.1 Insertion

Inserting a new data point, p_{ins}, to the dataset involves a dominance test and possibly a candidate reexamination if p_{ins} passes the dominance test. Dominance test in ZUpdate is based on Algorithm $Dominate$ (see Figure 7). Owing to the monotonic ordering property of Z-order curve (Property 3), p_{ins} can only be dominated by those skyline points whose Z-addresses are less than that of p_{ins}. Therefore, only a portion of $\mathcal{SL}$ (i.e., data points with Z-addresses smaller than p_{ins}'s) is examined. If p_{ins} passes the dominance test, p_{ins} is inserted to $\mathcal{SL}$ and the candidate reexamination is performed to discard those skyline points dominated by p_{ins}. Again, because of the monotonic ordering property, reexaminations occur

on those skyline points in $\mathcal{SL}$ with their Z-addresses larger than p_{ins}'s.



(a) Insertion of p_{insert} (b) Deletion of p_3 and p_5

Figure 10: Update of Skyline Results

Figure 10(a) shows a new data point p_{insert} that locates between p_3 and p_5 on the Z-order curve. In the dominance test, p_{insert} is compared against both p_1 and an RZ-region formed by p_2 and p_3. It passes the dominance test and hence is inserted to $\mathcal{SL}$. During the candidate reexamination, p_5 and p_6 (whose Z-addresses are greater than p_{insert}'s) are reexamined to check if they are dominated by p_{insert}. Finally, p_5 is removed while p_6 is retained.

5.2 Deletion

Updating a skyline result due to deletion is more complicated than insertion. Deleting any existing skyline point, say p_{del}, may get some data points previously dominated by p_{del} promoted to skyline. Since only those data points *exclusively* dominated by p_{del} are promoted, an issue is how to locate them efficiently from the source dataset. Owing to the monotonic ordering property of Z-Order curve, those dominated by p_{del} should have Z-addresses greater than that of p_{del}. ZUpdate starts traversal of $\mathcal{SRC}$ from the Z-address of p_{del} and lookups the RZ-regions and data points dominated by p_{del}, but not other skyline points in $\mathcal{SL}$. The traversal follows the non-descending order of Z-addresses. New skyline points obtained from the search are inserted to $\mathcal{SL}$.

Algorithm $ZUpdate(\mathcal{SRC}, \mathcal{SL}, P_{del})$
Input. $\mathcal{SRC}$: ZBtree for source dataset;
$\mathcal{SL}$: ZBtree for skyline points (with already P_{del} removed);
P_{del}: a set of deleted skyline points;
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 $maxpt$ **or**
6. $Dominate(\mathcal{SL}, n$'s $minpt, n$'s $maxpt)$ **then**
7. goto line 3.
8. **if** n is a non-leaf node **then**
9. **forall** children node c of n **do**
10. s.push(c);
11. **else** /* n is leaf node */
12. **forall** children point c of n **do**
13. **if** $\exists p \in P_{del}, p \vdash c$ **and** not $Dominate(\mathcal{SL}, c, c)$ **then**
14. $\mathcal{SL}$.insert(c);
15. **output** $\mathcal{SL}$;
End.

Figure 11: Algorithm $ZUpdate$ **(supporting multiple deletions)**

To alleviate the average skyline update cost, multiple deletions can be accumulated and performed as a batch. For instance, even though p_3 and p_5 in Figure 10(b) have been deleted at different times, the search for promoted skyline candidates due to their deletions can be processed together at a later time. Figure 11 outlines the pseudo-code of ZUpdate algorithm for searching promoted skyline points and it is general enough to handle multiple deletions. We use P_{del} to contain all the deleted skyline points. The algorithm traverses $\mathcal{SRC}$ and checks RZ-regions and data points against data points in P_{del} and $\mathcal{SL}$. RZ-regions of $\mathcal{SL}$ can alleviate some comparison overheads. Only those points exclusively dominated by P_{del} are collected and inserted to $\mathcal{SL}$. The algorithm terminates after $\mathcal{SRC}$ is traversed.

6. K-ZSEARCH FOR K-DOMINANT SKYLINE QUERY

With a relaxed k-dominance condition, k-dominant skyline query retrieves a smaller and representative subset of the conventional skyline points. In this section, we propose k-ZSearch, also based on the ZBtree, to process k-dominant skyline queries. To address the challenging issue of *cyclic dominance*, k-ZSearch adopts a filter-and-reexamine approach. In the filtering phase, we remove all those k-dominated data points and retain possible skyline candidates, which may contain false hits. In the reexamination phase, all candidates are reexamined to eliminate false hits. Section 6.1 and Section 6.2 detail these two phases.

6.1 Filtering Phase

The only difference of the filtering phase between k-ZSearch and ZSearch is the dominance condition. k-ZSearch traverses $\mathcal{SRC}$, the ZBtree for the source dataset, once to filter out data points or RZ-regions that are k-dominated by some candidate skyline points and retrieves candidates. The candidate set for k-ZSearch may contain false results due to cyclic dominance.

Extended from the definition of dominance condition to k-dominance condition between RZ-regions, Theorem 1 describes the transitive k-dominant relationship and Lemma 2 is derived for efficient candidate filtering in k-ZSearch.

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 $p \vdash_k p''$.* □

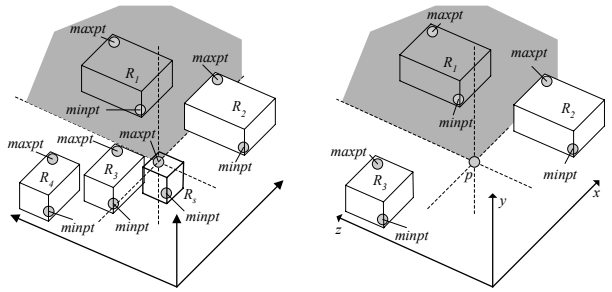
Proof. For (1), given certain k out of d dimensions, s_i, $p.s_i \leq p'.s_i \leq p''.s_i$ must hold, implying $p.s_i \leq p''.s_i$. Suppose there is one dimension out of k dimensions, s_j, such that $p.s_j < p'.s_j \leq p''.s_j$, in this case, $p.s_j < p''.s_j$. Hence, $p \vdash_k p''$.
Similarly for (2), given certain k out of d dimensions, s_i, $p.s_i \leq p'.s_i \leq p''.s_i$ 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 three cases hold:*

1. *if $maxpt(R) \vdash_k minpt(R')$, then $R \vdash_k R'$.*
2. *if $maxpt(R) \nvdash_k minpt(R') \wedge minpt(R) \vdash_k maxpt(R')$, then some points in R' may be k-dominated by others in R.*
3. *if $minpt(R) \nvdash_k maxpt(R')$, then $R \nvdash_k R'$.* □

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 data points in R'. By the second part of Theorem 1, all data points in R (except $maxpt(R)$) dominate $maxpt(R)$. As a result, $R \vdash_k R'$. As shown in Figure 12(a), R_1 and R_2 are 2-dominated by R_s in 3D space.
Case 2. $maxpt(R) \nvdash_k minpt(R') \wedge minpt(R) \vdash_k maxpt(R')$**.** Some points including $maxpt(R')$ might be k-dominated by some points in R. R_s and R_3 in Figure 12(a) form an example.
Case 3. $minpt(R) \nvdash_k maxpt(R')$**.** 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_4 in Figure 12(a) are an example for this case. ■

Lemma 2 is generic that an RZ-region can cover only one data point (see Figure 12(b)). Based on this, Algorithm k-$Dominate$ is



(a) RZ-regions (b) Point and RZ-region

Figure 12: 2-Dominance test in 3D space

derived to determine whether an RZ-region, R, presented as $minpt$ and $maxpt$, is k-dominated by any existing candidate k-dominant skyline point maintained by the ZBtree for k-$\mathcal{SL}$. The logic of this algorithm is outlined in Figure 13. It adopts breadth first search to traverse k-$\mathcal{SL}$. Similar to Algorithm $Dominate$, the search terminates when R is assured to be k-dominated by any RZ-region enclosing some candidate skyline points, or when all relevant nodes of k-$\mathcal{SL}$ are visited that indicates R not dominated.

```
Algorithm k-Dominate(k-SL, minpt, maxpt)
Input.   k-SL: ZBtree (k-dominant skyline points);
         minpt and maxpt: 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 children 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 children point c of n do
13.       if c ⊢_k minpt then
14.         output TRUE;
15.   output FALSE;
End.
```

Figure 13: Algorithm k-$Dominate$

With Algorithm k-Dominate, the filtering phase of k-ZSearch (line 1-18 in Figure 14) collects k-dominated candidate points in k-$\mathcal{SL}$. Meanwhile, those unqualified data points or index node entries (i.e., k-dominated by any existing 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 relaxed dominance condition. The filtering phase terminates when the $\mathcal{SRC}$ is completely traversed.

6.2 Reexamination Phase

The reexamination phase of k-ZSearch algorithm removes the false hits from candidates collected in the filtering phase. The main idea is that as long as a candidate in k-$\mathcal{SL}$ is found to be k-dominated by any other point either in k-$\mathcal{SL}$ or in T, it is removed from k-$\mathcal{SL}$ and inserted to T. All candidate points are reexamined. The logic of this phase is depicted in Line 19-26 in Figure 14.

Since the number of candidates maintained in k-$\mathcal{SL}$ is much smaller than that of points stored in T and those candidates usually have a greater dominating power, our reexamination checks every candidate data point 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 RZ-regions/data points in T k-dominate p, p retains in k-$\mathcal{SL}$. If some index nodes in T

```
Algorithm k-ZSearch(SRC)
Input.   SRC: ZBtree for source data set;
Local.   k-SL: 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-SL, n's minpt, n's maxpt) then
 6.       goto line 3. /* remove those dominated data points */
 7.     if k-Dominate(k-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 children node c of n do
12.         s.push(c);
13.     else /* leaf node */
14.       forall children point c of n do
15.         if k-Dominate(k-SL, c, c) then
16.           T.insert(c);            /* c for reexamination. */
17.         else
18.           k-SL.insert(c);         /* tentative result set */
      /*** re-examination phase ***/
19.   forall point p ∈ k-SL do
20.     if ∃p' ∈ k-SL, p' ≠ p ∧ p' ⊢_k p then
21.       remove p from k-SL;
22.       T.insert(p);
23.     else if ∃p'' ∈ T, p'' ⊢_k p then
24.       remove p from k-SL;
25.       T.insert(p);
26.   output k-SL;
End.
```

Figure 14: Algorithm k-$ZSearch$

need further exploring, they are replaced with all their children entries (either data points or index nodes) in T. Finally, those kept in k-$\mathcal{SL}$ are returned as the k-dominance skyline points.

7. PERFORMANCE EVALUATION

This section evaluates ZSearch, ZUpdate and k-ZSearch, respectively, by comparing them with the state-of-the-arts for processing skyline query, skyline update and k-dominant skyline query.

7.1 Experiment Settings

Our evaluations are based on both synthetic and real datasets. Synthetic datasets are generated based on *anti-correlated* distribution and *independent* distribution (described in Section 4.3). Correlated distribution is not included due to space limitation. The data dimensionality (d) varies from 4 to 16 and the data cardinality ranges from 10k to $10,000$k (ten millions) in order to evaluate the scalability of the proposed algorithms. Real datasets, namely, including NBA, HOU and FUEL follow anti-correlated, independent and correlated distributions, respectively[9]. 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, values of all datasets are normalized to $[0, 1023]^d$.

We implemented representative skyline algorithms, namely, SFS [7], BBS [13], BBS-Update [13], DeltaSky [17][10] and TSA [5] for comparison. Similar to most of the related works in the literature, we employ the *elapsed time* and the *I/O cost* as the main performance

[9]Those datasets are collected from `www.nba.com`, `www.ipums.org` and `www.fueleconomy.gov`, respectively.

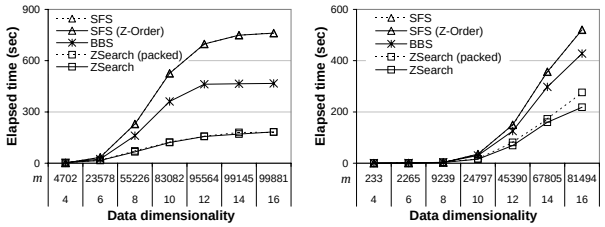
[10]Since DeltaSky focuses only on deletion, we adopt the insertion algorithm of BBS-Update for DeltaSky.

metrics. The former represents the duration from the time an algorithm starts to the time the result is completely returned. The latter is the number of disk pages accessed. We also measure the number of skyline points, m, and runtime memory used for data structures supporting different algorithms. All experiments were conducted on Linux 2.6.9 Servers with Intel Xeon 3.2GHz and 4GB 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 for storing skyline results and related data structures. Since Z-addresses can be used to derive the original attribute values, we only keep Z-addresses in ZBtree to support ZSearch, ZUpdate and k-ZSearch and derive the original attribute values as needed. The R*-tree adopted by BBS, BBS-Update and DeltaSky keeps data points in the leaf level. For SFS and TSA, records are sorted (by default) in accordance with the sum of all attribute values. All indices and sorted records are prepared prior to the experiments. ZBtree $\mathcal{SRC}$ is created by bulkloading. The results to be reported are the average performances of 30 sample runs unless specified otherwise.

7.2 Experiments on Skyline Search

The first set of experiments studies the performance of ZSearch in comparison with BBS and SFS, the state-of-art skyline search algorithms. Additionally, to study the potential advantages of Z-order on SFS, we employ Z-address to order data points for SFS (denoted as SFS (Z-Order)). In SFS (Z-Order), the additional key for sorting is not needed since Z-addresses can be used to recover the original attributes. As a result, the storage cost is smaller than SFS. We also implement a packed ZBtree (denoted as ZSearch (packed)) by filling all leaf nodes to their full capacity to examine the effect of our ZBtree node construction strategy that minimizes the sizes of RZ-Regions.

Effect of dimensionality. Figure 15 plots the elapsed time against the data dimensionality from 4 up to 16 for synthetic anti-correlated and independent datatsets. The number of skyline points (m) is plotted right below the x-axis. As observed in the figures, ZSearch performs the best while BBS outperforms SFS for both data distributions. SFS seriously suffers from intensive dominance tests that involve a lot of pairwise point-to-point comparisons. The performance deteriorates significantly as m increases. ZSearch and BBS respectively maintain $\mathcal{SL}$ and main-memory R-tree to reduce unnecessary dominance tests, especially effective when high data dimensionality is experimented. On the other hand, thanks for the RZ-region properties, ZSearch can determine whether a skyline point or an RZ-region is dominated at upper-level nodes of SL, resulting in shorter elapsed times than BBS that needs to reach the leaf nodes of the main memory R-tree every time. In other words, ZSearch can perform dominance tests much more effectively and hence reduce computational cost and improve the elapsed time performance.



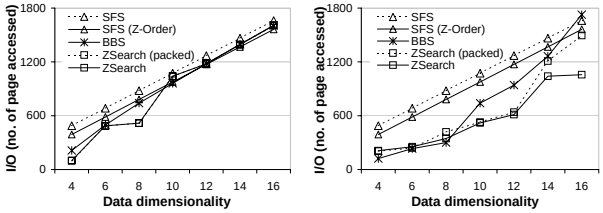
(a) Anti-correlated (b) Independent

Figure 15: Elapsed time (search, d:4-16, n:100k)

We further notice that the resulted curves of the two types of datasets are in different shapes. For anti-correlated datasets, many skyline points are resulted that incurs high computational overheads. When the dimensionality reaches 12, over 95% of data points are not dominated. As a result, the elapsed time performances for all algorithms reach the maximal and become flat. In contrast, for independent datasets with low data dimensionality, many data points (and index nodes) are dominated and thus pruned. Consequently, all the algorithms perform similarly well. As dimensionality increases, the performance of SFS deteriorates as we explained above. However, ZSearch grows in a moderate rate and performs better than BBS as discussed in Section 4.3.

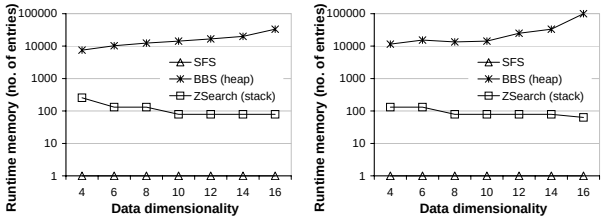
The experimental results also show no visible difference between SFS and SFS (Z-Order), indicating the sorting order is not a key factor for SFS algorithms. Meanwhile, ZSearch (packed) performs slightly worse than ZSearch for anti-correlated datasets, where almost all data points are not dominated and thus accessed eventually. As a result, our node construction strategy does not have a significant impact to the performance. On the other hand, for independent datasets, ZSearch outperforms ZSearch (Z-Order) as data dimensionality increases, demonstrating the effectiveness of our strategy. As many data points in independent datasets are dominated, fitting data points in RZ-regions can facilitate block based dominance test and efficient space pruning.

Figure 16 depicts the corresponding I/O costs. In general, ZSearch incurs the lowest I/O cost. For anti-correlated datasets, ZSearch slightly outperforms others in low data dimensionality (d:4-8). As dimensionality increases (d:10-16), all the algorithms result in similar I/O costs because the majority of points are not dominated and have to be retrieved from the source. For independent datasets, SFS still needs scanning the entire dataset, resulting in the highest I/O costs. BBS performs better than SFS in low dimensionality, but it loses its advantage when data dimensionality reaches 16 due to the curse of dimensionality. Compared with BBS, ZSearch incurs much fewer false hit pages, resulting in few I/O costs. SFS (Z-Order) requires less space than SFS and thus consistently results in less I/O cost than SFS. Z-Search (packed), although more compact in storage size than ZSearch, incurs false hits and thus results in a higher I/O cost than ZSearch.



(a) Anti-correlated (b) Independent

Figure 16: I/O cost (search, d:4-16, n:100k)



(a) Anti-correlated (b) Independent

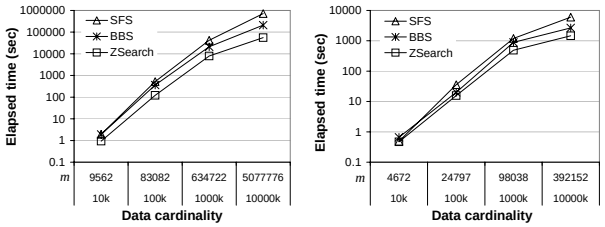
Figure 17: Runtime memory (search, d:4-16, n:100k)

Figure 17 depicts the runtime memory consumption in terms of the number of entries maintained in main memory to facilitate index traversal (in log scale)[11]. BBS uses a heap to order index pages and data points; and ZSearch uses a stack. Since SFS does not need any extra data structure, it incurs zero runtime memory consumption. BBS takes much more memory (up to 1000 times) than ZSearch especially when high dimensional datasets are searched. Moreover, BBS heap entry that maintains the distance to the origin of the space is slightly larger than ZSearch stack entry.

Effect of data cardinality. Figure 18 depicts the elapsed time

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

against the data cardinality (n: 10k up to 10000k). The number of skyline result points (m) is listed below the x-axis. The elapsed time of all algorithms (in log scale) increases as data cardinality grows. Among all, ZSearch produces the shortest elapsed time. This can be explained with similar reasons discussed above. Due to limited space, the results in terms of I/O cost and runtime memory consumption are not presented.



(a) Anticorrelation (b) Independent

Figure 18: Elapsed time (search, d:10, n:10k-10000k)

Dataset	m	SFS	BBS	ZSearch
NBA	10816	2.933, 228	3.364, 230	1.723, 131
HOU	5774	1.334, 874	2.169, 896	0.944, 346
FUEL	1	0.031, 164	0.001, 3	0.001, 3

Table 2: Real datasets (Skyline search) (time (msec), I/O)

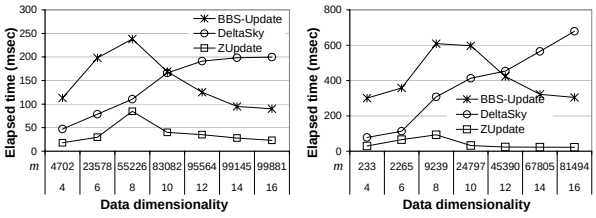
The experiment results of the real datasets are listed in Table 2. ZSearch clearly outperforms SFS and BBS, the state-of-the-art skyline search algorithms, for both the elapsed time (in msec) and the I/O costs. This shows the superiority of ZSearch for both real and synthetic datasets with various dimensionality and cardinality. In conclusion, ZSearch is the best skyline search algorithm.

7.3 Experiments on Skyline Result Update

The second set of experiments studies the performance of ZUpdate for skyline result update. We employ BBS-Update [13] and DeltaSky [17] as our comparison candidates. Recall that there are two types of update operations, namely *insertion* and *deletion*. To evaluate the performance for each type of operations, our experiment is conducted as follows. We first obtain an initial skyline result set. Then, among all skyline points, we randomly select 100 skyline points, delete them one by one from the source dataset/skyline result and update corresponding data structures. The average performance of 100 deletions is reported. Then, we insert the 100 deleted data points back one by one. These reinserted data points, i.e., some of initial skyline points, are not dominated by any other skyline points and they should dominate those skyline points promoted during deletions. We measure the performance and report the average performance of 100 insertions. For synthetic datasets, we vary the data dimensionality (d) from 4 up to 16 and data cardinality (n) from 10k up to 10,000k.

Figure 19 shows the performance of deletions for various data dimensionality (d:4-16), with a fixed cardinality (100k). The elapsed times of BBS-Update and ZUpdates increase when d is increased from 4 to 8. This is because many skyline points become not dominated and get promoted. Then, the performance gradually drops since the majority of data points are skyline points which leaves only a few or even no data points being checked and being promoted. DeltaSky keeps increasing as d grows due to a high scanning overheads for multiple sorted lists to identify exclusively dominated data points or index nodes. The length of each sorted lists equals the number of skyline points. Among all the three algorithms, ZUpdate performs the best. These findings are consistently observed in both anti-correlated and independent datasets.

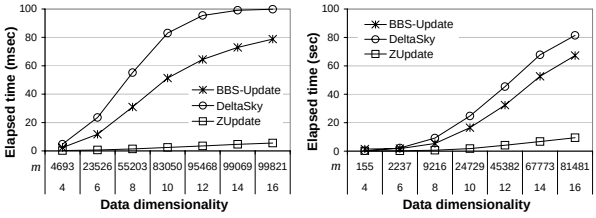
Figure 20 shows the performance of insertions. Compared with deletions, insertions are lightweight operations since they do not need to retrieve data points from the source datasets. In the figure, ZUpdate is the winner and BBS-Update performs better than DeltaSky. With $\mathcal{SL}$, dominance tests for newly inserted data points



(a) Anti-correlated (b) Independent

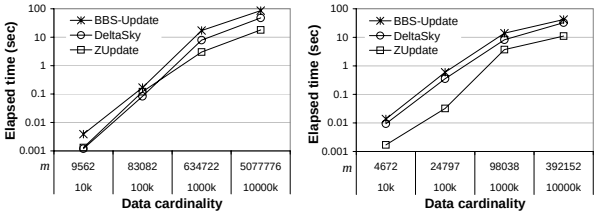
Figure 19: Elapsed time (deletion, d:4-16, n:100k)

can be done at a higher node level (and thus in larger blocks) but BBS-Update needs to go down to the leaf level. In addition, BBS-Update incurs expensive R-tree update costs. DeltaSky also suffers a lot from updating both multiple sorted lists and a main memory R-tree, making it less competitive than BBS-Update. This experiment shows that ZUpdate is good at handling skyline result update for insertions. Figure 21 shows the performance of deletions of the three algorithms for various data cardinality (10k-10,000k), with dimensionality fixed at 10. Again, ZUpdate consistently outperforms the others. We omitted the results for insertions to save space.



(a) Anti-correlated (b) Independent

Figure 20: Elapsed time (insertion, d:4-16, n:100k)



(a) Anti-correlated (b) Independent

Figure 21: Elapsed time (deletion, d:10, n:10k-10000k)

Next, we evaluate the update performance on the real datasets. The results are listed in Table 3. The observations are consistent to what we made from synthetic datasets. Again, for both deletion and insertion, ZUpdate outperforms BBS-Update and DeltaSky, the known best skyline update algorithms.

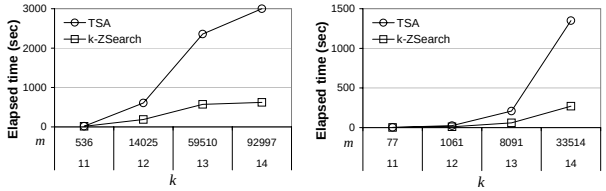
7.4 Experiments on k-Dominant Skyline Search

The third experiment set examines the performance of k-ZSearch algorithm, for k-dominant skyline search. In this part, we compare k-ZSearch against TSA [5], an efficient k-dominant skyline search algorithm. We generate a synthetic dataset with 15 as the dimensionality and evaluate the performance of both TSA and k-ZSearch with respect to different k's and cardinality. In our experiments, when k is smaller than 11, only a few or even no results are returned. Therefore, we vary k from 11 to 14 (out of 15).

Figure 22 and Figure 23 depict the experiment results. The corresponding numbers of k-dominated skyline points are listed below the x-axis. In terms of elapse time (as shown in Figure 22), k-ZSearch outperforms TSA significantly. The advantage of k-ZSearch decreases as k is reduced (e.g., $k = 11$), because only a very small number of k-dominated skyline points are obtained when k is small. When a large k is evaluated, k-ZSearch effectively prunes the search space and requires only one dataset access (while TSA needs two full scans). Therefore, I/O costs incurred

Dataset	m	BBS-Update		DeltaSky		ZUpdate	
		del	ins	del	ins	del	ins
NBA	10816	78.37	4.18	42.25	5.09	14.21	1.27
HOU	5774	492.11	5.22	482.31	5.98	339.96	2.44
FUEL	1	0.10	0.001	0.15	0.008	0.10	0.001

Table 3: Real dataset (skyline result update) (time (msec))



(a) Anti-correlated (b) Independent

Figure 22: Elapsed time (k-Dominant, k:11-14, n:100k)

by k-ZSearch is less than 50% of that of TSA. The plot is omitted to save space. Figure 23 shows the experiment results with different data cardinalities (with k fixed at 12). Again, k-ZSearch outperforms TSA and the difference becomes more obvious as cardinality increases, because TSA has to access the dataset twice but only a small portion of data points are actually the skyline results. Table 4 shows the performance based on the real datasets. We set k to $(d-1)$, $(d-2)$, and $(d-3)$ with d equivalent to the dataset dimensionality. Consistent to our expectation, k-ZSearch is better than TSA, showing the superiority of k-ZSearch for k-dominant skyline search.

8. CONCLUSION

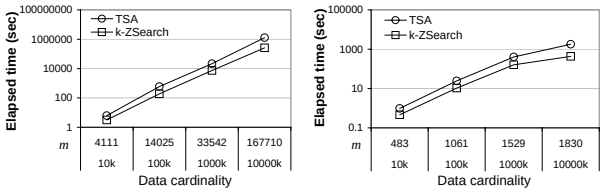
In this paper, we analyze the skyline problems and exploit the ordering and clustering properties of the Z-order curve which match perfectly well with the skyline processing strategies. Through this study, we have made the following contributions:

1. We propose to use ZBtree as a fundamental mechanism to coherently support various skyline processing algorithms including ZSearch, ZUpdate and k-ZSearch.
2. The ZSearch algorithm scales very well in both dimensionality and cardinality, and soundly outperforms BBS [13] and SFS [7], the state-of-the-art search algorithms.
3. The ZUpdate algorithm efficiently updates skyline results in presence of insertions and deletions in the underlying dataset. It soundly outperforms BBS-Update [13] and DeltaSky [17], the best skyline update algorithms available today.
4. The k-ZSearch algorithm efficiently handles the cyclic dominance problem in k-dominant skyline query and soundly outperforms TSA [5], the representative k-dominant skyline search algorithm.
5. We conduct an extensive performance evaluation to compare our proposal with existing best algorithms designed specifically for certain problem domains (i.e., skyline queries, skyline updates and k-dominance skyline queries). The result demonstrates the superiority of our algorithms and validates the design principles and basis of our proposal.

The potential of Z-order curves in tackling skyline processing problems has not been fully exploited. As the next steps, we are investigating other Z-order properties to support other variants of skyline queries.

9. ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their constructive comments, which improved the presentation of this paper. Special thanks go to Prof. Bernhard Seeger at University of Marburg, who provided many helpful suggestions in the revision of this paper. This research is supported in part by the National Science Foundation under Grant no. IIS-0328881 and IIS-0534343.



(a) Anti-correlated (b) Independent

Figure 23: Elapsed time (k-Dominant, k:12, n:10k-10000k)

Dataset	k	m	TSA	k-ZSearch
NBA	12	3794	7.931	2.696
	11	682	1.980	0.731
	10	79	0.322	0.171
HOU	5	22	0.815	0.226
	4	0	0.487	0.220
FUEL	5	1	0.063	0.001
	4	1	0.062	0.001

Table 4: Real dataset (k-Dominant Skyline) (time (sec))

10. REFERENCES

[1] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient Distributed Skylining for Web Information Systems. In *EDBT*, pages 256–273, 2004.

[2] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When Is "Nearest Neighbor" Meaningful? In *ICDT*, pages 217–235, 1999.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, pages 421–430, 2001.

[4] C. Buchta. On the Average Number of Maxima in a Set of Vectors. *Information Processing Letter*, 33(2):63–65, 1989.

[5] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding K-Dominant Skylines in High Dimensional Space. In *SIGMOD*, pages 503–514, 2006.

[6] S. Chaudhuri, N. N. Dalvi, and R. Kaushik. Robust Cardinality and Cost Estimation for Skyline Operator. In *ICDE*, page 64, 2006.

[7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In *ICDE*, pages 717–816, 2003.

[8] P. Godfrey, R. Shipley, and J. Gryz. Maximal Vector Computation in Large Data Sets. In *VLDB*, pages 229–240, 2005.

[9] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline Queries Against Mobile Lightweight Devices in MANETs. In *ICDE*, page 66, 2006.

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

[11] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *ICDE*, pages 502–513, 2005.

[12] J. A. Orenstein and T. H. Merrett. A Class of Data Structures for Associative Searching. In *PODS*, pages 181 – 190, 1984.

[13] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline Computation in Database Systems. *ACM TODS*, 30(1):41–82, 2005.

[14] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the UB-Tree into a Database System Kernel. In *VLDB*, pages 263–272, 2000.

[15] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *VLDB*, pages 301–310, 2001.

[16] Y. Tao and D. Papadias. Maintaining Sliding Window Skylines on Data Streams. *IEEE TKDE*, 18(2):377–391, 2006.

[17] P. Wu, D. Agrawal, O. Egeciglu, and A. E. Abbadi. DeltaSky: Optimal Maintenance of Skyline Deletions without Exclusive Dominance Region Generation. In *ICDE*, pages 486–495, 2007.

[18] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing Skyline Queries for Scalable Distribution. In *EDBT*, pages 112–130, 2006.