Scalable Skyline Computation Using Object-based Space Partitioning

Shiming Zhang, Nikos Mamoulis and David W. Cheung
Department of Computer Science
The University of Hong Kong, Hong Kong
{smzhang, nikos, dcheung}@cs.hku.hk

ABSTRACT

The skyline operator returns from a set of multi-dimensional objects a subset of superior objects that are not dominated by others. This operation is considered very important in multi-objective analysis of large datasets. Although a large number of skyline methods have been proposed, the majority of them focuses on minimizing the I/O cost. However, in high dimensional spaces, the problem can easily become CPU-bound due to the large number of computations required for comparing objects with current skyline points while scanning the database. Based on this observation, we propose a dynamic indexing technique for skyline points that can be integrated into state-of-the-art sort-based skyline algorithms to boost their computational performance. The new indexing and dominance checking approach is supported by a theoretical analysis, while our experiments show that it scales well with the input size and dimensionality not only because unnecessary dominance checks are avoided but also because it allows efficient dominance checking with the help of bitwise operations.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Experimentation, Performance, Theory

Keywords

skyline, preference, space partitioning

1. INTRODUCTION

The *skyline query*, as an efficient tool for preference-based data analysis [27, 13], has attracted a lot of attention in the database community. It has a wide range of real applications [8, 26] and it can easily be incorporated into commercial database systems, as SQL is being extended with clauses for the support of preference queries [3, 10, 19, 8].

Specifically, given a set of d-dimensional objects $\mathcal{O}$, a skyline query retrieves all *superior* objects, which can not be dominated by any others in $\mathcal{O}$ with respect to user preferences. Here, an object o dominates another object o', if and only if o is not worse than o' in all dimensions and better than o' in at least one dimension (assuming that dimensional domains can be partially or totally ordered according to some preference or quality criterion).

Skyline queries are useful in multi-criteria decision making applications that involve high dimensional and large datasets, especially for clients with a limited bandwidth connection since they only return a minimum of superior candidates from the server. For example, consider a person using a wireless client device (e.g., a smart mobile phone) to connect to a phone shopping website (e.g., http://www.phonescoop.com/phones/finder.php) and look for a cell phone. He/she may care about a wide range of features including weight, size, standby time, screen size, etc. and wish to manually select a phone among the best ones in the market. Computing and returning the skyline over the concerned features, filters out a large number of inferior cell phones, hopefully leaving only a manageable number of phones ferried to the client device for manual evaluation.

Several algorithms have been proposed targeting the efficient skyline evaluation on large datasets. There is a number of methods that operate on pre-computed indexes on the data, including Bitmap [23], Index [23], NN [14], BBS [19], LS [18] and ZSearch [15]. Techniques that do not rely on indexes include BNL [3], D&C [3], SFS [10], LESS [11], and SaLSa [1]. Intuitively, the index-based schemes are faster than index-independent strategies, since they avoid accessing the entire data collection, yet their applicability is significantly limited by the indexing requirement. First, in many applications the data are dynamically produced (e.g., they arrive from a stream or they are output from an ad-hoc database operator), so they may not be indexed. Second, the skyline may have to be computed for an ad-hoc set of features (dimensions), some of which may be dynamically derived (e.g., distance to a reference location); we may not assume that multi-dimensional indexes pre-exist for any possible set of (static or dynamic) preference features. Finally, multi-dimensional indexes like R-trees have their own limitations as they suffer from the well-known curse of dimensionality. In addition, in high-dimensional problems, index-based skyline methods face memory management problems as they need to manage a huge number of skyline points [31, 6, 7].

We observe that skyline computation in high dimensional spaces is challenging and CPU-intensive pairwise dominance tests (as opposed to I/O) become the dominant cost factor. In specific, the state-of-the-art approaches that do not depend on indexing (e.g., [10, 11, 1]), first sort the data, such that no point dominates a previous one in the order. Then, while scanning the sorted file, they compare each accessed point with the skyline points found so far; these are kept in a memory buffer. If an accessed point is not dom-

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

inated by the previously found skyline points, it is added to the current skyline. The bottleneck of this process is the comparison of each accessed point with the current skyline, which can become as large as the available memory.

In this work, we propose an **object-based space partitioning** (OSP) scheme, which recursively divides the d-dimensional space into 2^d separate partitions w.r.t. a reference skyline object, and facilitates progressive skyline retrieval on high dimensional datasets. Using this scheme, our method organizes the current skyline points in a search tree, which facilitates efficient skyline computation; every accessed point is compared only to a small number of current skyline points, using the tree to guide search. We perform a theoretical analysis, which estimates the expected number of comparisons that have to be performed in order to decide whether an accessed point is in the skyline. In addition, we encode the partitions using bitmaps and use a **left-child/right-sibling** (LCRS) tree to organize them. The benefit of this tree is that it allows efficient breadth-first search, while the encoding allows for fast bitwise comparisons. Our experimental evaluation demonstrates that our algorithms are orders of magnitude faster than the current state-of-the-art, practically minimizing the required comparisons during skyline retrieval.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 introduces our OSP scheme and analyzes its pruning power for dominance tests. Thereafter, we present the LCRS tree implementation of our recursive partitioning scheme and show the details of three skyline algorithms based on it; two memory-based and one that is suitable for cases where the skyline does not fit in memory. Section 4 describes how the LCRS skyline tree is updated when data are inserted or deleted. Section 5 presents our experimental evaluation on both real and synthetic datasets. In Section 6 we discuss how our method can be (i) extended for k-dominant skyline queries, (ii) adapted for parallel skyline evaluation, and (iii) applied for skyline computation on partially ordered domains. Finally, Section 7 concludes the paper.

2. RELATED WORK

The skyline query is a popular and powerful paradigm for extracting interesting objects from multi-dimensional databases. The problem, also known as maximal vector computation [2], was first coined into the database community in [3]. It is related to other classic problems in theory, such as convex hull [2] and Pareto optimality [13]. Early solutions, based on the *divide & conquer* (D&C) paradigm, do not scale well for large datasets, since they do not take into account main memory limitations. [3] adapts the basic D&C approach to operate with external memory. The main idea is to recursively divide the dataset into partitions that fit in memory and compute the local skyline for each of them. The global skyline is computed by progressively merging the local ones based on a bushy merge tree. As argued in [3, 10, 11], the average performance of D&C deteriorates with the dimensionality of the problem, because as dimensionality increases the non-skyline objects have greater chance to belong to the local skyline of their partition and the sizes of local skylines grow significantly.

[3] also introduces a *block nested loop* (BNL) algorithm, which scans the dataset while using a memory buffer for tracking the objects not dominated by others. If the buffer overflows, then space is freed by flushing some objects to a temporary disk file. After the first pass, objects that entered the buffer before any other object was written to the temporary file are guaranteed to be skyline objects. The rest are kept in the buffer and the temporary file is used as input for a new run of BNL. The algorithm may require a large number of passes until the complete skyline is computed. The *sort filter skyline* (SFS) [10] is an improvement of BNL, which first sorts the dataset topologically with the help of a monotone function (e.g., sum of coordinates, assuming they have been normalized). Sorting guarantees that each object cannot be dominated by ones that follow it in the order. As a result, each object that is pushed into the buffer window can immediately be reported as part of the skyline. The number of passes over the data is then equal to the size of the skyline over the size of the memory buffer. An optimized version of SFS, called *linear elimination sort for skyline* (LESS), is proposed in [11]. LESS uses a small buffer, called *elimination-filter window* in the initial pass of the external sort routine of SFS, which keeps a small set of objects used to prune others dominated by them early. Further, LESS combines the last pass of the external sort in SFS with the first filter-scan of SFS (i.e., first pass of the BNL component of SFS). In SFS and LESS, all objects should be scanned at least once after sorting. *Sort and limit skyline algorithm* (SaLSa) [1] strives to avoid scanning the complete set of sorted objects. First, the authors suggest an optimal sorting function, which orders the points according to their minimum coordinate value among all dimensions. Second, during the filter-scan process, this method checks whether all points in the remaining dataset are dominated by a so-called *stop* object that can be determined in $O(1)$ time from the data accessed so far. However, the performance of this method is drastically affected by the data distribution and increasing dimensionality; in high-dimensional problems, the pruning power of the stop object is limited. All sort-based techniques (SFS, LESS, SaLSa) suffer from the large number of computations required during the filter-scan step, as every read point should be compared with the skyline points in the buffer.

The aforementioned algorithms do not require that the input data have been indexed. On the other hand, a set of other techniques [3, 23, 14, 19, 18, 15] exploit data indexes to accelerate skyline queries. [3] first proposed simple algorithms that use B-trees or R-trees for skyline evaluation. Then, two progressive processing methods, *Bitmap* and *Index*, were proposed in [23]. *Bitmap* encodes all data into a bitmap structure so that the skyline can be identified quickly by a bitwise *and* operation. *Index* partitions the entire data into several lists, indexes each list by a B-tree and uses the trees to find the local skylines, which are then merged to a global one. [14] observes that the *nearest neighbor* (NN) object to the origin must be in the skyline and uses an R-tree to find the NN, and then segments the remaining data into overlapping partitions based on the NN. The next nearest neighbors are then iteratively found in each partition. Multiple traversals of the R-tree are required to remove duplicates at the overlapping regions. The *branch and bound skyline* (BBS) algorithm introduced in [19] avoids these pitfalls by prioritizing accesses at partially dominated nodes of the R-tree. The algorithm is shown to be I/O optimal and superior to the method of [14]. [15] proposed a ZBtree that encodes and clusters all objects with the help of a Z-order curve, which is compatible with the dominance relation. As we discussed in the introduction, index-based approaches have certain limitations that make them useful only for special cases.

Recently, some studies [5, 4, 8, 18, 28, 27] went beyond skyline evaluation for totally ordered numerical domains and consider partially ordered domains involving categorical or nominal dimensions. Most of them adopt a partial-to-total domain mapping mechanism and then apply existing total order methods, which however suffer from the complex and large size of partially ordered domains [5, 13]. Finally, a *lattice skyline* (LS) algorithm, introduced in [18], uses a lattice structure to answer skyline queries with dimensions drawn from low-cardinality domains. This method becomes inefficient if the number or size of the domains is large and is not applicable if more than one high-cardinality domain is present. In

this paper, our focus is on totally ordered domains of high cardinality. In Section 6, we discuss how our methods can be adapted for partially ordered domains.

To tackle the curse of dimensionality, several proposals extended or adapted the definition of skyline in order to consider dimensional subspaces in the dominance relationships between objects [31, 24, 7, 17, 6]. In addition, a top-k query that considers dominance relationships was proposed in [30]. Moreover, efforts have been devoted to dynamic skyline search [9, 21], probabilistic skyline computation [20, 16, 12], skyline computation over uncertain data, and skyline queries over data streams [22]. Skyline queries have also been studied in metric spaces [9], or parallel [29, 25] and distributed [26] environments.

3. SKYLINE PROCESSING USING OSP

Our objective is to improve the performance of sort-based skyline algorithms (e.g., [1, 10, 11]). Recall that these algorithms topologically pre-sort the data based on a monotone function, which requires that if object o precedes o' in the order, then o' cannot dominate o. The sorted dataset is scanned; if an object is not dominated by any other that precedes it then it is guaranteed to be in the skyline. Thus, each accessed object is compared with all skyline points found so far and potentially added to the skyline set. If there is no space in memory to fit the new skyline object, then it is written to a temporary file. After the first pass, the skyline objects in the buffer are reported and the algorithm is repeated for the temporary file. Our goal is to minimize the computational cost of testing whether the currently read object is in the skyline, during scans.

For this purpose, this section first introduces our Object-based Space Partitioning (OSP) scheme and defines partition-wise dominance and other associated properties. A detailed analysis estimates the pruning power of our OSP strategy in dominance tests; that is the expected number of skyline objects in the buffer compared with the currently accessed point. An efficient tree implementation of our recursive OSP scheme is then introduced, which facilitates partition-wise dominance checking. Thereafter, three skyline algorithms for skyline processing based on OSP are presented.

Without loss of generality, we assume that lower values have higher preference in all dimensions; i.e., object o dominates object o', denoted by $o \succ o'$, iff $\forall i \in [1,d], o_i \leq o'_i$ and $\exists j \in [1,d], o_j < o'_j$. Table 1 shows the frequently used notation in the paper.

Symbol	Interpretation
$\mathcal{O}, o$	object-set, object
d	dimensionality
$\mathcal{H}_i^{o+}$ ($\mathcal{H}_i^{o-}$)	Superior (inferior) halfspace w.r.t. obj. o and dim. i
$A_o(V)$	d-bit address of partition V w.r.t. obj. o
$\mathcal{L}_o(o')$	Locating partition of o' w.r.t. obj. o
$\mathrm{D}_o(o')$	partitions that dominate or equal $\mathcal{L}_o(o')$
$\mathrm{U}_o(o')$	partitions that are dominated by $\mathcal{L}_o(o')$
m	partitioning tree depth
β	max. number of objects in partition tree leaves

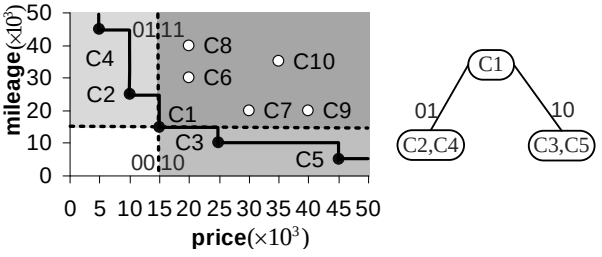
Table 1: Notation

3.1 Object-based Space Partitioning (OSP)

Consider a set $\mathcal{O}$ of objects in a d-dimensional space $\mathcal{R}^d$. Given an object $o = \{o_1, o_2, \ldots, o_d\}$ in $\mathcal{O}$, for any dimension $i \in [1,d]$, $\mathcal{R}^d$ can be divided into two halfspaces, *Superior Halfspace* $\mathcal{H}_i^{o+}$ and *Inferior Halfspace* $\mathcal{H}_i^{o-}$, by the hyperplane $R_i = o_i$, such that $\forall o' \in \mathcal{H}_i^{o+}, o'_i < o_i$ and $\forall o' \in \mathcal{H}_i^{o-}, o'_i \geq o_i$. Therefore, the space $\mathcal{R}^d$ can be divided into 2^d partitions using o and any object $o' \in \mathcal{O} \backslash o$ must lie in explicitly one of them. Formally:

DEFINITION 1. *Given a skyline object o in $\mathcal{R}^d$, namely the **reference**, o divides $\mathcal{R}^d$ into 2^d separate partitions by d hyperplanes $\{R_1, R_2, \ldots, R_d\}$, where $R_i = o_i$, and the **address** of any partition V, is a d-bit number $A_o(V)$, such that for every dimension i, $A_o(V)[i]$=0, iff $V \subset \mathcal{H}_i^{o+}$; $A_o(V)[i]$=1, iff $V \subset \mathcal{H}_i^{o-}$. Furthermore, the partition that contains an object $o' \in \mathcal{O} \backslash o$ is called the **locating partition** of o' w.r.t. o and denoted by $\mathcal{L}_o(o')$.*

Based on Definition 1 and given a reference skyline object o, we can define a disjoint partitioning of the space into 2^d regions, and give them addresses from 0 to $2^d - 1$. In practice, the partition address of any object o' w.r.t. o can be directly computed by comparing its coordinates with those of o. Since o is a skyline point, the partition with address $00\ldots0$ must be empty. In addition, all objects in the partition with address $11\ldots1$ (besides o's duplicates) must be dominated by o, thus they are not skyline points. Finally, the objects in all other partitions are incomparable with the reference object o. Therefore, in practice, we only need to consider the partitions with addresses in $[1,\cdots,2^d\text{–}2]$. These 2^d–2 partitions are organized by a tree, as depicted in Figure 1(b) for the data of Figure 1(a). In this 2D example, where attributes are price and milage in a database with used cars, the reference skyline object is C1; objects C6–C10 are dominated by C1 and are therefore pruned (they are in partition with address 11). Objects C2 and C4 are incomparable to C1 and in partition with address 01, whereas objects C3 and C5 are also incomparable to C1 and in partition with address 10.



(a) OSP (b) The Partition Tree

Figure 1: An example on Object-based Space Partitioning

We now define dominance between partitions V, W which belong to the same OSP (see Definition 1), as follows.

DEFINITION 2. *If partitions V and W belong to the same OSP $\mathcal{S}$ based on o, V dominates W, denoted by $V \succ W$, iff $\forall i \in [1,d]$, $A_o(V)[i] \leq A_o(W)[i] \wedge \exists j \in [1,d], A_o(V)[j] < A_o(W)[j]$. Otherwise, V does not dominate W, denoted by $V \not\succ W$. Further, V and W are incomparable iff $V \not\succ W \wedge W \not\succ V$.*

For example, in a 3D space OSP, the partition with address 001 dominates the partition with address 011 and it is incomparable with partition 100. We write $V \succeq W$ to represent that $V \succ W$ or $V = W$. Immediately, the conclusions below can be derived from the above definitions.

THEOREM 1. *Given $o, o', o'' \in \mathcal{O}$, if $o' \succ o''$, then $\mathcal{L}_o(o') \succeq \mathcal{L}_o(o'')$.*

PROOF. $o' \succ o'' \Rightarrow o'_i \leq o''_i$ in every dimension i. So, in dimension i and w.r.t. o, o' and o'' must either be (i) in the same halfspace ($o', o'' \in \mathcal{H}_i^{o+}$ or $o', o'' \in \mathcal{H}_i^{o-}$) or (ii) in different halfspaces: $o' \in \mathcal{H}_i^{o+}$ and $o'' \in \mathcal{H}_i^{o-}$). Case (i) implies $A_o(o')[i] = A_o(o'')[i]$ and case (ii) implies $A_o(o')[i] < A_o(o'')[i]$. Therefore, $A_o(o')[i] \leq A_o(o'')[i]$ holds in any dimension i. Hence, $A_o(\mathcal{L}_o(o'))[i] \leq A_o(\mathcal{L}_o(o''))[i]$ in any dimension i and $\mathcal{L}_o(o') \succeq \mathcal{L}_o(o'')$. □

LEMMA 1. *$V \succ W$, iff (i) $A_o(V) < A_o(W)$ and (ii) $(A_o(V) \mid A_o(W)) = A_o(W)$.*

PROOF. Condition (ii) implies that $\forall i \in [1,d], A_o(V)[i] \leq A_o(W)[i]$ and condition (i) implies $\exists j \in [1,d], A_o(V)[j] < A_o(W)[j]$. So, $V \succ W$ and the inverse derivation is obvious. □

COROLLARY 1. *V is incomparable to W, iff $(A_o(V) \,|\, A_o(W)) > max\{A_o(V), A_o(W)\}$.*

PROOF. From bit-wise operation principles, we know that $(A_o(V) \,|\, A_o(W)) \geq max\{A_o(V), A_o(W)\}$, so we should prove that V is incomparable to W, iff $(A_o(V) \,|\, A_o(W)) \neq max\{A_o(V), A_o(W)\}$. This holds due to Lemma 1. □

LEMMA 2. *If V and W are incomparable, then $\forall v \in V, \forall w \in W$, v and w are incomparable.*

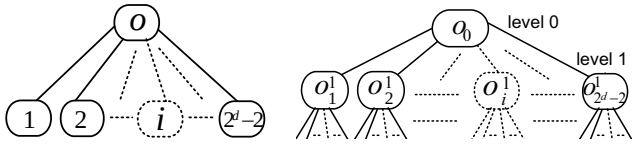
PROOF. If V and W are incomparable then $V \not\succ W \wedge W \not\succ V$. Let $v \in V, w \in W$. $V \not\succ W \Rightarrow \exists i \in [1,d], A_o(V)[i] > A_o(W)[i] \Rightarrow \exists i \in [1,d], A_o(v)[i] > A_o(w)[i] \Rightarrow w_i < o_i < v_i \Rightarrow v \not\succ w$. Symmetrically, we can show that $W \not\succ V \Rightarrow \exists i \in [1,d], v_i < o_i < w_i \Rightarrow w \not\succ v$. Hence, v and w are incomparable. □

Lemma 2 implies that the pairwise dominance tests among incomparable partitions can be safely ignored. Conversely, any object o' may be dominated by some objects in the dominating partitions of $\mathcal{L}_o(o')$ (including $\mathcal{L}_o(o')$), as indicated by Theorem 1, and the pairwise dominance tests for objects in them against o' are necessary, if we wish to check whether o' is a skyline object.

DEFINITION 3. *The **dominating partition set** $\mathrm{D}_o(o')$ and the **dominated partition set** $\mathrm{U}_o(o')$ of any object $o' \in \mathcal{O} \backslash o$ w.r.t. o are defined as:*

$$\mathrm{D}_o(o') = \{V \in \mathcal{S} \mid V \succeq \mathcal{L}_o(o')\}$$
$$\mathrm{U}_o(o') = \{V \in \mathcal{S} \mid \mathcal{L}_o(o') \succ V\}$$

Suppose that the object o' is not worse than o in k dimensions, or equivalently $A_o(o')$ (or $\mathcal{L}_o(o')$) holds k 1-bits. Then, $|\mathrm{D}_o(o')| = 2^k - 1$ and $|\mathrm{U}_o(o')| = 2^{d-k} - 1$. If we have organized the already found skyline points in an OSP w.r.t. o, then to determine if a candidate $o_c \in \mathcal{O}$ that is currently accessed is in the skyline, it is sufficient that it passes all dominance tests with skyline objects in the partitions of $\mathrm{D}_o(o_c)$. Lemma 1 inspires a sequential accessing order to facilitate progressive partition-wise dominance tests against the candidates and safely skip all incomparable partitions according to Corollary 1. The skyline objects in the partitions of $\mathrm{D}_o(o_c)$ are expected to be much fewer than the overall number of skyline points found so far, therefore our method is expected to have big computational savings over previous skyline techniques that compare the accessed objects with the complete skyline set in the buffer.

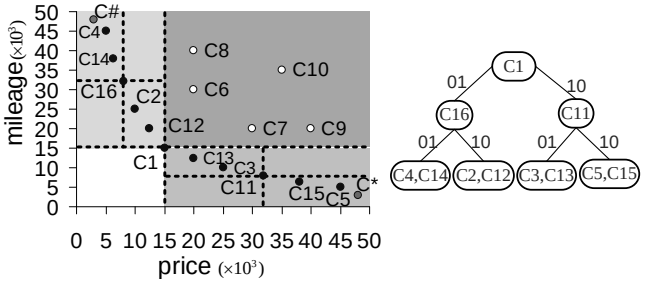


(a) OSP Skyline Tree (b) ROSP Skyline Tree

Figure 2: Object-based Space Partitioning Skyline Trees

Recursive Object-based Space Partitioning (ROSP) Definition 1 gives us the basic object-based partitioning scheme to divide the space $\mathcal{R}^d$ into 2^d non-overlapping partitions w.r.t. a skyline reference o. Any object $o' \in \mathcal{O}$ can then be identified to lie in exactly one of these partitions. The $2^d - 2$ considered partitions are indexed by a partition tree, as shown in Figure 2(a). In fact, this OSP scheme w.r.t. a skyline object can be recursively applied to each partition and all the properties and derivations presented above hold for each subsequent partitioning. The partitions can be organized by a hierarchical partition tree, as shown in Figure 2(b). Specifically, each internal node contains the *reference* skyline object, which subdivides the partition represented by this node. For any object o', we can iteratively compare it with the reference object in the partition where it belongs at each level of the tree and ultimately find the finest-level partition where it belongs. During this tree traversal, for each node, corresponding to a reference object o, we need to check whether o' passes all dominance tests over its dominating partitions $\mathrm{D}_o(o')$. If o' is not pruned during this process and finally reaches a leaf locating partition, we can be sure that o' belongs to the skyline of $\mathcal{O}$. We call this partition tree *skyline tree* and use it to hierarchically index all skyline objects.

As an example of this recursive OSP scheme, consider the points shown in Figure 3(a). Assume that object C1 is used at the level-0 partitioning. Partitions 01 and 10 by C1, are divided again, using the skyline objects C16 and C11, respectively. The corresponding ROSP partition skyline tree is shown in Figure 3(b). Candidate skyline object C* is first compared with C1 and found to belong to partition 10. Then it is compared with C11 and sent to its subpartition 10 containing C5 and C15. After being compared with these objects, it is found to be a new skyline object and it is inserted in that partition. Similarly, C# is compared to C1, C16, and finally leaf objects C4, C14. In this 2D example, note that a candidate skyline object is compared to only one object per-tree level, until it reaches a leaf partition, where it is compared with all objects contained there. In higher dimensionality cases, an object o' may have to be compared with multiple partitions per level, as its dominating partition set $\mathrm{D}_o(o')$ has greater size than one. This means that multiple paths of the skyline tree are traversed in general. In the next section, we theoretically estimate the number of required comparisons for uniform data distributions.



(a) ROSP (b) ROSP Skyline Tree

Figure 3: Recursive Object-based Space Partitioning Example

3.2 Partition-wise Dominance Tests Estimate

For simplicity, assume that there is a large number of objects in the d-dimensional space $\mathcal{R}^d$, uniformly and independently distributed (UI assumption). This implies that if the space is partitioned by a skyline object o which is on or near enough to the main diagonal, each indexed partition over the OSP partition skyline tree, as shown in Figure 2(a), contains the same number of skyline objects. In other words, a new skyline candidate o_c falls in any one of these partitions with the same probability $1/(2^d - 2)$. Based on Theorem 1, it is sufficient that o_c passes all dominance tests with objects in the partitions in $\mathrm{D}_o(o_c)$, in order to verify if o_c is in the skyline. The expected ratio of existing skyline points to be compared with the current candidate o_c is given by the following theorem.

THEOREM 2. *Under UI assumption and with the help of an OSP w.r.t. skyline object o, the average dominance test ratio $R(d)$ for a skyline candidate o_c, is $O(\frac{3^d - 2^{d+1} + 1}{(2^d - 2)^2}) < O((\frac{3}{4})^d)$.*

PROOF. Assuming that $A_o(o_c)$ holds k 1-bits, where $k \in [1, d-1]$, we have $|\mathrm{D}_o(o_c)| = 2^k - 1$. There are C_d^k partitions whose addresses contain exactly k 1-bits. So, the average number of fetched partitions for o_c from the total of 2^d-2 partitions is $\sum_{k=1}^{d-1} \frac{1}{2^d-2}(2^k-1)C_d^k$. According to the *binomial theorem* in math, $\sum_{k=0}^{d} C_d^k 2^k = 3^d$ and $\sum_{k=0}^{d} C_d^k 1^k = 2^d$. Then $\sum_{k=1}^{d-1} \frac{1}{2^d-2} C_d^k (2^k - 1) = \frac{1}{2^d-2}(\sum_{k=0}^{d} C_d^k 2^k - \sum_{k=0}^{d} C_d^k 1^k - 2^d + 1) = \frac{3^d - 2^{d+1}+1}{2^d-2}$. Therefore, the average dominance test ratio $R(d)$ for o_c is, $R(d) = \frac{1}{(2^d-2)} \sum_{k=0}^{d-1} \frac{1}{2^d-2} 2^k C_d^k = \frac{3^d-2^{d+1}+1}{(2^d-2)^2} < O((\frac{3}{4})^d)$ $\square$

Now suppose that a recursive partitioning scheme is used instead of OSP and the existing skyline objects are indexed with the help of a ROSP skyline tree. Suppose that the depth of recursive partitioning of the space $\mathcal{R}^d$ is m (that is, the tree has m levels). Each non-leaf partition still has the same average dominance test ratio $R(d)$ for a skyline candidate among its $2^d - 2$ sub-partitions. In other words, there are about $R(d)^i(2^d-2)^i$ skyline objects to compare with the candidate at level i. Among all skyline objects in a partition at the leaf level, there are about $\sum_{i=0}^{m-2} R(d)^i(2^d-2)^i + \beta R(d)^{m-1}(2^d-2)^{m-1}$ ones to compare with the candidate, where β is the size of the leaf partition. Therefore, the fraction of fetched objects in this partition is $\frac{\sum_{i=0}^{m-1} R(d)^i(2^d-2)^i + \beta R(d)^{m-1}(2^d-2)^{m-1}}{\sum_{i=0}^{m-1}(2^d-2)^i + \beta(2^d-2)^{m-1}} = O(R(d)^{m-1})$.

THEOREM 3. *Under UI assumption and using a ROSP skyline tree with height m, the average pairwise dominance test ratio $R(*)$ for any skyline candidate is $O(R(d)^m) < O((\frac{3}{4})^{dm})$.*

PROOF. Immediate, as ROSP search is equivalent to performing dominance tests using a simple OSP, where each partition has a fraction $O(R(d)^{m-1})$ of the total skyline points. Hence, $R(*) = O(R(d)^m) < O((\frac{3}{4})^{dm})$. $\square$

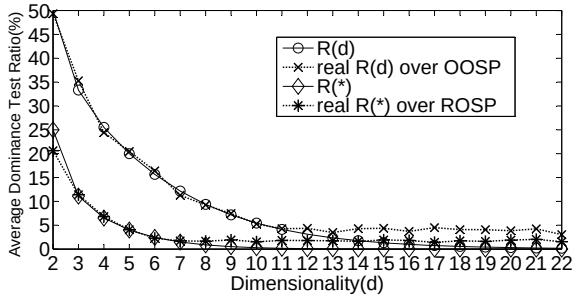


Figure 4: Average Dominance Test Ratio with $d, m = 3$

Figure 4 evaluates our analysis, by showing the average dominance test ratio together with the actual results obtained on uniform datasets with 1M+1K skyline objects. Each presented real result is the average value of the recorded ratios for the remaining 1K skyline objects, after indexing 1M objects using a skyline tree (OSP or ROSP tree). Different values of the problem dimensionality are tested. This graph illustrates the accuracy of our analysis and shows the great pruning power of managing current skyline points using an OSP or ROSP tree. Observe that the percentage of skyline objects that are fetched for dominance tests against the candidate reduces when d increases. In addition, the ROSP scheme performs better than the basic OSP tree, as expected from our analysis. For instance, the dominance tests against a candidate involve only about 50% of the indexed objects when $d = 2$, about 35% when $d = 3$, and less than 10% when $d = 8$, in the OSP scheme. If $d = 8$ and we use a ROSP tree with three levels, the savings in comparisons are huge (98%).

In the worst case, the skyline candidate o_c is always in a partition whose address holds $(d-1)$ 1-bits and 2^{d-1} partitions should be checked; in this case the ratio is $\frac{2^{d-1}}{2^d-1} \approx 0.5$ for a basic OSP tree, and 0.5^m if we use a ROSP tree. Obviously, this is still much better than comparing all current skyline objects.

The above analytical results are derived under the UI assumption for points in $\mathcal{R}^d$. In practice, the data may be anticorrelated or skewed and most of the indexed partitions may be empty, resulting in an unbalanced indexing tree. This is why the ratio for the experimental results in Figure 4 is greater than the ratio in the analysis, in high dimensional spaces; when $d \geq 10$, the skyline objects are not evenly distributed in all partitions. Obviously, the choice of the reference objects in the partitioning greatly impacts the tree structure and the performance. In the worst case, the ROSP tree degenerates to a list and then our dominance test strategy is similar to the exhaustive dominance test method used by previous techniques (e.g., LESS [11]). To overcome this problem, we employ a random guessing strategy for the reference object in each partitioning. Additionally, we use a *Left-Child/Right-Sibling* tree structure for implementing the ROSP tree, which facilitates efficient dominance tests.

3.3 Left-Child/Right-Sibling Skyline Tree

As shown in Figure 5, the skyline objects can be indexed by a *left-child/right-sibling* (LCRS) tree based on their partition addresses w.r.t. the ROSP scheme. This implementation serves two purposes: (i) high space efficiency, since many partitions are empty and need not be indexed, and (ii) efficient sequential accessing of partitions and objects in a breadth-first fashion.

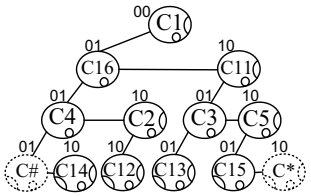


Figure 5: A LCRS tree for the data of Figure 3(a)

DEFINITION 4. *A node e in the LCRS tree is a 4-tuple $<$**o**, **pa,child,sb**$>$, where **o** is a skyline object (which may subdivide this node partition if the node is not a leaf); **pa** is the address of the partition represented by this node; **child** is a pointer to the non-empty leftmost sub-partition; **sb** is a pointer to the right (i.e., next) non-empty sibling partition.*

Obviously, the root node is the first reference object to partition $\mathcal{R}^d$, so its pa should be 0 and sb is $null$. Sibling nodes ordered in ascending partition address pa. Figure 5 illustrates a LCRS tree for the data of Figure 3(a). Note that in this example, the tree has four levels (as opposed to the exemplary ROSP tree of Figure 3(b), which has only 3 levels), because we apply recursive partitioning until there is only one object remaining at each partition.

As for any node e in the LCRS tree, all objects in its child sub-tree together with its node object $e.o$ are in the same partition and share the same partition address w.r.t. its parent node object $\hat{o}$. Consider a candidate skyline object o with address $A_{\hat{o}}(o)$. If $(A_{\hat{o}}(o) \mid e.pa) = e.pa$, it is clear that the partition represented by node e is in $\mathrm{D}_{\hat{o}}(o)$ and $e.pa \leq A_{\hat{o}}(o)$. Therefore, objects in the sub-tree of e including its reference object $e.o$ should be compared with the candidate o to detect whether o is dominated by any of them. Otherwise, node e will be directly skipped over and all objects in its child sub-tree together with this node object $e.o$ will be

ignored. Therefore, the dominance tests for any candidate o can be easily implemented based on a preorder tree traversal of the LCRS tree, as we discuss in detail in the next section.

The LCRS tree can easily be adapted to a LCRS *partitioning* tree, if some leaf nodes e, except from their reference object $e.o$, also keep all the objects in the corresponding partition (without recursively re-partitioning them further). For example, the children of C16 in Figure 5 could be leaf nodes that contain all objects in the corresponding partitions (i.e., {C4,C#,C14} and {C2,C12}, respectively, like the ROSP tree of Figure 3(b)). Such an adaptation is used by our algorithms which are presented in the remaining of this section.

3.4 LCRS Tree Growth

In this section, we show how to manage the skyline objects dynamically with the help of a LCRS tree. We describe in detail how candidate points are compared with the existing ones in the tree and how the tree grows as new skyline points are discovered.

Algorithm 1: PreOrderDominate($e, o, pa, is\mathcal{L}$)

Input: e: LCRS tree node; o: candidate; pa: $\mathcal{L}(o)$'s address; $is\mathcal{L}$: whether e is on o's locating path
Output: whether o is dominated by some skyline point

```
1  begin
2    if (e.pa | pa) = pa then                          ▷ check dominating node
3      Pa := A_{e.o}(o)                                ▷ compute A_{e.o}(o)
4      if Pa = 2^d − 1 then                            ▷ o is in the partition 11···1
5        return true                                   ▷ e.o ≻ o
6      inSL := isL ∧ (e.pa = pa)                       ▷ whether o definitely is in e
7      if e.child ∧ e.child.pa ≤ Pa then
8        if PreOrderDominate(e.child, o, Pa, inSL) then
9          return true                                 ▷ o is dominated by a descendant of e
10     else if inSL then                               ▷ insert o as e.child
11       e.child := <o, Pa, null, e.child>
12       return false
13   if e.sb ∧ (e.sb.pa ≤ pa) then
14     if PreOrderDominate(e.sb, o, pa, isL) then
15       return true                                   ▷ o is dominated by a sibling of e
16   else if isL ∧ (e.pa < pa) then                    ▷ insert o as e.sb
17     e.sb := <o, pa, null, e.sb>
18   return false
19 end
```

As discussed before, every node e in the tree should be accessed and compared with the candidate o if its partition address $e.pa$ w.r.t. its parent node object $\hat{o}$ satisfies two conditions: (i) $e.pa \leq A_{\hat{o}}(o)$ and (ii) $(A_{\hat{o}}(o) \mid e.pa) = e.pa$. Therefore, for a candidate object o, we perform a preorder traversal to the tree, as described by Algorithm 1. Whether a sub-tree of e or one of e's sibling nodes needs to be accessed depends on the two conditions. In specific, for each visiting node e there are two cases against the candidate o with the locating partition address pa w.r.t. e's parent node object $\hat{o}$.

- Case 1 (lines 2∼12): $(e.pa \mid pa) = pa$. In this case, node e is a dominating partition node for the candidate o. The algorithm must first check whether this node's object $e.o$ dominates o by computing o's partition address $\mathcal{P}a$ w.r.t. $e.o$ (line 2). If o is not pruned (line 5), then the algorithm is recursively run for $e.child$. Here, the method first checks whether node e is in the locating partition of o, indicated by variable $in\mathcal{SL}$ together with $is\mathcal{L}$ (line 6). $is\mathcal{L}$ indicates whether the visiting node e is in the locating path (path from root to the final locating node) inherited from the recursive calling at the parent. If $is\mathcal{L} = false$, we perform dominance tests for o with e and $e.child$, but o cannot be inserted into the sub-tree of e (in such case, the recursion on $e.child$ (line 8) will return $false$). Otherwise, i.e., $is\mathcal{L} = true$, e is o's locating partition (i.e., $in\mathcal{SL} = true$) and $e.child.pa$ is smaller than $o.pa$, the algorithm is recursively called for $e.child$ (line 8) and possibly o will be inserted in that sub-tree. However, if e does not have a child with $e.child.pa < o.pa$, then o will be inserted to the tree as the leftmost child of e and linked to that child with a sibling pointer (lines 10∼12).
- Case 2 (lines 13∼18): $(e.pa \mid pa) \neq pa$. In this case, node e is skipped over and control is passed to its sibling $e.sb$, if a sibling $e.sb$ exists and satisfies condition (i). Otherwise, the sibling is created dynamically and o is inserted there (line 17).

This preorder traversal is not terminated until the candidate o is (i) dominated by object or (ii) inserted into e as a new skyline object. The special variable $is\mathcal{L}$ is used to indicate whether the candidate o is in a node that has the same address as o w.r.t. its parent. If so, the candidate can be inserted under that node. Otherwise, the node is skipped.

To comprehend the functionality of PreOrderDominate, consider the LCRS tree of Figure 5 and consider candidate object C*. First, C* is compared with the root object C1 and we attempt to insert it to its child C16, since in the root run of the algorithm C*.pa=C1.pa=00 and C1 does not dominate C*. C16 is incomparable with C* (Case 2), so C16 is skipped over and control is passed to its sibling C11. C11 is in the same partition as C*, w.r.t. C1 (Case 1). So, these two are compared and C*'s address w.r.t. C11 is computed. C11 is in the locating path of C*, so we call the procedure for the child of C11, that is C3 (line 8). Continuing in this fashion, C* eventually reaches C15, and since they are incomparable and there are no more siblings of C15 to check, C* is inserted as C15's sibling and the algorithm terminates, having decided that C* is a new skyline object. Similarly, the candidate C# is compared successively with the root object C1, C16, and eventually reaches C4. Since C4 and C# are incomparable, in this recursion the algorithm will reach lines $10 \sim 12$ because for the child C14 of C4, we have C14.pa $= 10 >$C#.pa $= 01$. Therefore, C# is inserted as a leftmost the child of node C4 and the original child of node C14 will be linked as the sibling of C#; then, the algorithm terminates, having decided that C# is a new skyline object.

3.5 OSP Skyline Algorithms

In this section, we propose two algorithms that are based on the LCRS tree and the growth procedure described in Section 3.4. When describing these two algorithms, we assume that the skyline is small enough to fit in memory. The case where the skyline grows larger than the available memory is discussed in Section 3.7.

The first method, **OSPSOnSortingFirst** (Algorithm 2), is a straightforward application of the LCRS tree growth procedure. It follows the sort-based paradigm of the state-of-the-art skyline algorithms [10, 11, 1]. The data are first topologically sorted, such that each object cannot be dominated by all objects behind it in the ordered dataset. We then have to set a skyline object as the root of the tree; the first object in order is guaranteed to be in the skyline. In order to increase the chances for balanced level-0 partitions, instead of selecting the first object as reference, we may choose a random skyline point, by performing a scan over the data. We investigate the effectiveness of such an approach in the Section 5. Then, OSPSOnSortingFirst scans the remaining objects and applies the PreOrderDominate procedure with $pa = 0$ and $is\mathcal{L} = true$. Finally, $\mathcal{SL}$ is reported as the skyline tree.

Our second method, **OSPSOnPartitioningFirst**, does not rely on sorting, but attempts to partition the dataset and solve indepen-

Algorithm 2: OSPSOnSortingFirst($\mathcal{O}$)

Input: $\mathcal{O}$: dataset;
Output: $\mathcal{SL}$: $LCRS$ Skyline Tree

```
1  begin
2    Sort O by a topological monotone function F
3    s_first := a skyline object in O
4    SL :=< s_first, 0, null, null >                    ▷initialize SL
5    foreach o ∈ O\s_first do                  ▷ check all other objects in O
6      PreOderDominate(SL, o, 0, true)
7    return SL
8  end
```

dent problems, hinted by Theorem 1 and Lemma 1. Specifically, the main idea is to dynamically partition the dataset $\mathcal{O}$, while growing the LCRS tree. Algorithm 3, which is the pseudocode of our second method, is an adapted version of PreOrderDominate, which was presented in Section 3.4.

We know that the objects in a partition with a larger address cannot dominate objects in ones with smaller address. So, if we grow the tree starting with the partitions with the smaller addresses, we can use it to effectively prune the partitions dominated by them, as the process continues. The details of this method are as follows. First (lines 2∼8), we recursively divide the current leftmost child partition $\mathcal{SO}$ of a dynamically defined LCRS partition tree[1] until it contains no more than a maximum number of objects (β in line 3 of Algorithm 3). Then, we compute the local skyline tree $\mathcal{SL}$ for the objects in $\mathcal{SO}$. Thereafter, $\mathcal{SL}$ filters all partitions $\mathrm{U}_o(\mathcal{SO})$ dominated by it among all current sibling nodes (line 9). Finally, we recursively call the algorithm by setting as $\mathcal{SO}$ its sibling, or backtrack the recursion tree to the next partition in order if there are no more siblings. While partitioning $\mathcal{SO}$ at line 7, the algorithm prunes all dominated objects by the reference r. Note that the reference object r at the partitioning of $\mathcal{SO}$ should be a skyline object. To find such an object, it suffices to search only within the partition (a linear scan is required), since points there dominated by other partitions should have been filtered earlier (as we will explain shortly).

Algorithm 3: OSPSOnPartitioningFirst($\mathcal{SO}$)

Input: $\mathcal{SO}$: LCRS Partition Tree
Output: $\mathcal{SL}$: $LCRS$ Skyline Tree

```
1  begin
2    if SO = null ∨ |SO.O| = 0 then return null
3    if |SO.O| ≤ β then
4      return OSPSOnSortingFirst(SO.O)
5    r :=a skyline object in SO.O
6    SL :=< r, SO.pa, null, null >
7    partition SO w.r.t. r, prune all r's dominating objects
8    SL.child :=OSPSOnPartitioningFirst(SO.child)
9    FilterDominatedPartitions(SO.sb, SL)
10   SO := SO.sb
11   SL.sb :=OSPSOnPartitioningFirst(SO)
12   return SL
13 end
```

If the leaf node capacity β in line 3 is set to 1, the partitions are recursively subdivided until one object is contained and that object is definitely as a skyline object. In general, smaller β results in deep partitioning, which helps to prune more data and reduce the cost compared to sorting. On the other hand, a small β incurs more partitioning that can increase the I/O cost.

[1]The LCRS partition tree is similar to the LCRS skyline tree, but each leaf node contains a set of objects from $\mathcal{O}$ instead of a skyline object o.

The filtering process of line 9 is implemented by Algorithm 4. If the tested sibling partition $\mathcal{SO}$ is dominated by the root node of $\mathcal{SL}$, the objects in it will be filtered by $\mathcal{SL}$ using PreOderDominate with $is\mathcal{L} = false$ (lines 3∼6), since we only need to apply dominance tests over $\mathcal{SL}$, but the surviving objects may not be inserted into $\mathcal{SL}$. The method recursively accesses the sibling partitions of $\mathcal{SO}$ in order, checking all dominated partitions (line 7). If a partition becomes empty, then it will be discarded in order to avoid redundant nodes in the partition tree (lines 8∼9). After the completion of this filtering process, any objects dominated by some skyline objects in $\mathcal{SL}$ will be discarded in all dominated partitions which are siblings of $\mathcal{SO}$. Observe that the current local skyline $\mathcal{SL}$ will not be processed again until we backtrack to the sibling partitions of its parent.

Algorithm 4: FilterDominatedPartitions($\mathcal{SO}, \mathcal{SL}$)

Input: $\mathcal{SO}$: $LCRS$ partition; $\mathcal{SL}$: skyline tree pre-sibling of $\mathcal{SO}$
Output: $\mathcal{SO}$: $LCRS$ partition filtered using $\mathcal{SL}$

```
1  begin
2    if SO = null then return
3    if (SO.pa | SL.pa) = SO.pa then                    ▷SL ⪰ SO
4      foreach o ∈ SO.O do
5        if PreOderDominate(SL, o, SO.pa, false) then
6          remove o from SO.O
7    FilterDominatedPartitions(SO.sb, SL)
8    if |SO.O| = 0 then
9      delete SO from LCRS partition tree; SO := SO.sb
10 end
```

As an example, consider a 3D dataset $\mathcal{O} = \{o_0, o_1, \cdots, o_9\}$. Assume that β as 3. First, the algorithm will pick a skyline object r from $\mathcal{O}$; assume that $r = o_0$. Using r, the algorithm partitions the dataset into three subsets: P001=$\{o_1, o_3, o_4\}$, P011=$\{o_5, o_6\}$, and P101=$\{o_2, o_7, o_8, o_9\}$, with addresses 001, 011, and 101, respectively. Partition P001 will be processed first and since its size is not greater than β, its skyline tree will be directly computed using the OSPSOnSortingFirst algorithm (Algorithm 2). Assume that the returned local skyline tree $\mathcal{S}_{001}$ consists of o_1, o_3, o_4, as shown in the 2nd step of Figure 6. In the next step, we filter the sibling partitions P011 and P101 of P001, using $\mathcal{S}_{001}$, since both of them are dominated by partition P001. Assume that none of the objects in them can be discarded. In the next step, we continue processing the next sibling node P011 using the same process and return the local skyline tree $\mathcal{S}_{011}$ containing o_5 and o_6. This tree is linked as sibling of $\mathcal{S}_{001}$. We then invoke the filtering process, but there is nothing to be done by it, as P011 does not dominate P101. In the last step, we process partition P101. Since it contains more than β objects, it is re-partitioned using o_2 into three partitions containing o_7, o_8, and o_9, respectively. After processing these three partitions, the algorithm will backtrack to the root and terminate, returning the skyline tree. Figure 6 shows how the skyline tree grows progressively as Algorithm 4 is applied on this dataset.

OSPSOnPartitioningFirst can be faster than OSPSOnSortingFirst because it avoids sorting and in addition it examines the data in a more principled order, attempting to prune objects as soon as skyline points are found in their dominating partitions.

3.6 Discussion

In this section, we first discuss how the proposed algorithms can seamlessly incorporate the early termination optimization of SaLSa [1] (called *limiting*). Then, we analyze their space requirements and expected processing time.

In our LCRS Skyline Tree, we can keep two additional minimal max-coordinate values in each node e (used by SaLSa[1]) among

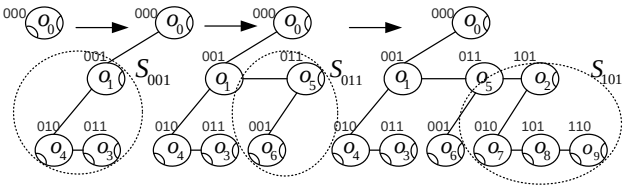


Figure 6: LCRSSTree growing over OSPSOnPartitioningFirst

all skyline objects in (i) the sub-tree rooted at e and (ii) e's sibling subtree, denoted by T_e and $T_{e.sb}$, respectively. Hence, the stop condition of SaLSa can be applied to each node using both T_e and $T_{e.sb}$. If the stop condition with T_e is satisfied when visiting node e, the dominance tests in (i) can be safely omitted. Similarly comparing to e's sibling subtree can be skipped using $T_{e.sb}$. By testing the termination condition on the root node of the LCRS Skyline Tree, OSPSOnSortingFirst can terminate, as early as possible. OSPSOnPartitioningFirst applies the same idea in each partition and when compares partitions dominating each other.

During search, only the current LCRS skyline tree $\mathcal{SL}$ must be kept in memory for both algorithms. In addition, the depth of the recursive calling in partition-wise dominance tests is bounded by the maximum path length, which is bounded by the size of the skyline r. Hence, the space complexity of the algorithms is $O(r)$. For an arbitrary candidate o_c, if it is not a skyline object, it must be dominated by one skyline object among $\mathrm{D}_o(o_c)$ at each level of the LCRS skyline tree $\mathcal{SL}$. Therefore, the expected number of dominance tests for it is $O(R(*)|\mathcal{SL}|)$ under UI. In the worst case, where $\mathcal{SL}$ degrades to a linked list, the time complexity of our methods is the same as that of previous approaches, i.e., $O(nr)$, where n is the size of the dataset. However, as discussed in Section 3.2, we expect the number of dominance tests to be much lower, namely $O(n \cdot R(*) \log r)$, if the tree is balanced.

3.7 OSPS on Bounded Memory

A closer look on OSPSOnPartitioningFirst reveals that, during the execution of the algorithm, we can report and remove from memory the part of the computed local skyline tree that has already been used to filter objects in the partitions that it dominates. In this section, we devise an external skyline algorithm based on this idea to handle the case where the skyline tree grows larger than the available system memory. The details of this extended version of OSPSOnPartitioningFirst are given by Algorithm 5.

Here, a temporary file $\mathcal{T}$ is utilized for gradually collecting the local skyline tree nodes swapped out of the memory. If the examined partition does not fit in memory, it is re-partitioned using one of its skyline objects and the algorithm is recursively applied to its leftmost subpartition (lines 15∼17). Otherwise, OSPSOnPartitioningFirst is applied to return the skyline tree $\mathcal{SL}$ only involving the processed partition $\mathcal{SO}$. Obviously, $\mathcal{SL}$ is a subtree of the overall skyline tree and can be directly written to the result file $\mathcal{T}$ (line 7). The algorithm traces back the path from the root that leads to $\mathcal{SL}$ (using $\mathcal{P}arent$ pointers) and uses $\mathcal{SL}$ to filter all the partitions that are siblings to any node in this path (lines $9 \sim 12$). The key difference from OSPSOnPartitioningFirst is in this filtering strategy. Recall that OSPSOnPartitioningFirst only filters the dominated partitions among the siblings of the processed partition $\mathcal{SO}$. Here, $\mathcal{SL}$ first filters all its dominated sibling partitions (line 8). Then, the parent node object and the reference object which divides the parent partition are pushed into $\mathcal{SL}$ (line 10). This dynamically grown skyline tree $\mathcal{SL}$ filters all dominated partitions at the parent level (line 11), and progressively checks all partitions that are dominated

Algorithm 5: OSPSOnOverflowingMemory($\mathcal{SO}$)

```
   Input: SO: LCRSPTree
   Output: T: file where skyline is written
1  begin
2    if SO = null then return
3    if |SO.O| ≤ β then                          ▷SO.O fits in memory
4      Cur := SO, Parent := SO.parent, SO := SO.sb
5      Cur.sb := null                            ▷retrieve SL only in Cur
6      SL :=OSPSOnSortingFirst(Cur)
7      output SL to file T                       ▷store local skyline tree
8      FilterDominatedPartitions(SO, SL)
9      while Parent do      ▷go up to filter ancestral siblings using SL
10       SL :=< Parent.o, Parent.pa, SL, null >
11       FilterDominatedPartitions(Parent.sb, SL)
12       Parent := Parent.parent
13     remove SL from memory
14   else                                  ▷recursively partition overflowing SO
15     r :=a skyline object in SO.O
16     partition SO w.r.t. r, prune all r's dominating objects in SO and
       output r into temporary file T
17     OSPSOnOverflowingMemory(SO.child)
18   OSPSOnOverflowingMemory(SO)
19 end
```

by nodes in the path that links the root with $\mathcal{SL}$. Then, $\mathcal{SL}$ can be removed from memory (line 13) to make space for the next partition, as the skyline objects in $\mathcal{SL}$ have already been used to filter any possible object in $\mathcal{O}$ that they can dominate. Having finished with $\mathcal{SL}$, the recursion will continue to process the next sibling of the current partition (line 18) or backtrack to the sibling of its parent. When all partitions are processed, the final skyline tree will have been collected into the skyline file $\mathcal{T}$.

4. SKYLINE UPDATES

The skyline may change due to subsequent updates to the database, and hence should be incrementally maintained to avoid re-evaluation from scratch. The LCRS skyline tree facilitates such skyline updates efficiently. In this section, we discuss how insertions and deletions are handled by an existing skyline tree. An insertion to the dataset may cause the deletion of some skyline objects which are dominated by the new object. Similarly, a deletion of a skyline object may result in new objects becoming part of the skyline.

4.1 Insertion

When a new object o_{int} is inserted, the first thing to do is to traverse the skyline tree $\mathcal{SL}$ to see if it is dominated by an existing skyline object. If o_{int} cannot be pruned, then it should be inserted into $\mathcal{SL}$. However, in this case, o_{int} may dominate some objects on the existing skyline which now have to be removed. To accomplish both tasks, we can easily extend the PreOderDominate algorithm (Algorithm 1) as follows. First, we perform preorder traversal as in the original algorithm until we find out that o_{int} can be pruned or insert o_{int} as a new skyline object. In the second case, we traverse the tree upwards from the new leaf that contains o_{int} and identify the partitions at each level, which are dominated by o_{int}. Skyline objects in these partitions are accessed (with the help of the tree) and any object found to be dominated by o_{int} is deleted from the tree. If a deleted object is in a non-leaf tree node, instead of explicitly deleting the node (requiring expensive tree re-organization), we can simply mark it as "non-skyline" object and not use it for filtering (but only for re-direction) in subsequent uses of the tree.

4.2 Deletion

The deletion of an existing object $o_{del} \in \mathcal{O}$ is handled as follows.

If o_{del} is not a skyline object, no further processing is required. Otherwise, we access the objects dominated by o_{del} in topological sort order and apply the PreOderDominate algorithm (Algorithm 1) for each of them using the existing tree (we remove o_{del} from the tree prior to this).

5. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the performance of the proposed algorithms by comparing them with LESS [11] and SaLSa [1] as well as ZUpdate [15], using both synthetic and real datasets. All algorithms in Table 2 were implemented in C++ and all experiments were conducted on a Linux 2.6.22 Server with Intel Xeon 2.50GHz CPU and 8GB RAM.

Algorithm	Description
LESS	Linear Elimination Sort for Skyline [11]
SaLSa	Sort and Limit Skyline algorithm [1]
OSPSSF	OSP Skyline using Sorting First (Sec. 3.5)
OSPSSF-1st	OSPSSF using first skyline object (Sec. 3.5)
OSPSSF-lim	Variant of OSPSSF with limiting (Sec. 3.6)
OSPSPF	OSP Skyline on Partitioning First (Sec. 3.5)
OSPSPF-lim	Variant of OSPSSF with limiting (Sec. 3.6)
OSPSOM	OSP Skyline on Bounded Memory (Sec. 3.7)
ZUpdate	Skyline Update in Z order [15]
OSPSInsert	OSP Skyline Update at Insertion (Sec. 4.1)
OSPSDelete	OSP Skyline Update at Deletion (Sec. 4.2)

Table 2: Description of the algorithms

5.1 Experiment Settings

Three types of synthetic datasets, *anti-correlated* (AC), *uniform and independent* (UI) and *correlated* (CO) distributions, are generated to model different scenarios according to the methodology in [3]. Due to the space limitation, some results on CO are not reported.[2] The data dimensionality (d) varies from 2 to 22 and the data cardinality (n) ranges from 10K to 1M to evaluate the scalability of the proposed algorithms. All dimensions are totally ordered domains which are normalized to a $[0, 1000]^d$ space. Accordingly, three real datasets are adopted in our evaluation, denoted by NBA, Household, and Color,[3] which follow AC, UI and CO distributions, respectively. NBA contains 19,181 statistics from regular seasons during 1946–2008, each of which corresponds to the statistics of an NBA player's performance in 21 aspects (such as points scored, rebounds, assists, field goals made, etc). However, we only consider 10 ones among 21 statistics in our evaluation since others may be missing in some records (e.g., steals were not recorded before 1973). Household consists of 127,931 data, each representing the percentage of an American family's annual expenses on 6 types of expenditures (e.g., electricity, gas, phone, etc). Color is a 9-dimensional dataset containing 68,040 objects, each representing the first three moments of the RGB color distribution of an image.

Three monotone functions, *Sum* (the sum of all coordinates), *Entropy* [11] and *minC* [1], are used for sorting the datasets in the different algorithms. LESS uses *Entropy*, SaLSa uses *minC*, and our sort-based method uses *Sum* by default. We use an EF window that fits 200 objects in the first sorting pass of LESS in our experiments. In addition, both LESS and SaLSa (with backward strategy [1]) are always completed in a single pass, when sufficient memory can be available for sorting and storing the results, and the buffer size in OSPSOM, β, is set to 10,000 objects by default, but to 1 object in OSPSPF. A ZBtree [15] is created using bulk loading to be tested for skyline updates. Our methods for insertion and deletion are adapted to work on the ZBtree (i.e., for searching all dominated objects by the deleted skyline object). All the settings are similar to those of ZUpdate in [15] on the evaluation of skyline updates. All results reported are the average performance over 20 iterations unless specified otherwise.

[2]Results on CO are similar to those on UI and AC.

[3]These datasets are collected from www.nba.com, www.ipums.org, and kdd.ics.uci.edu, respectively.

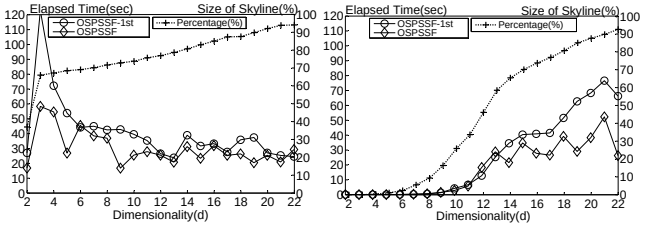
5.2 Experimental Results on Synthetic Datasets

5.2.1 The Effect of Guessing First Skyline Object

We first investigate the difference of using the first skyline object in the sort-order as the first partitioning object in our sort-based method OSPSSF (described in Section 3.5). We compare two versions of the algorithm; OSPSSF, where we use a random skyline object for the top partitioning, and OSPSSF-1st, where we use the first object in the topological sort order (all data are sorted by *Sum* during execution). A random skyline point can be identified if we randomly pick an object o in $\mathcal{O}$ and then scan the ordered file backwards from the position of o until the beginning of the file. If o is found to be dominated by a predecessor o', then we set $o = o'$ and continue the comparisons backwards from this point.

Figure 7 plots the elapsed time against the data dimensionality from 2 to 22 among 100K objects for OSPSSF and OSPSSF-1st. The percentage of skyline objects in the result is plotted in the same figures, but indicated by the right y-axis. In most cases, using a random object leads to an improvement in the performance of OSPSSF in both AC and UI datasets (similar results hold for CO). In addition, we found out that the performance of OSPSSF-1st is greatly impacted by the sorting function, whereas using a random skyline object in OSPSSF is more robust. On the other hand, finding a random skyline object has additional overhead, as it may require an additional scan over the data, while it makes no difference on UI and CO datasets of low dimensionality. In summary, choosing the level-0 partitioning skyline object carefully pays off only for anti-correlated data or high dimensional problems.

Interestingly, the elapsed time does not grow with the increasing size of the skyline and dimensionality in AC. It fluctuates and becomes almost flat in high dimensional datasets, where there are more skyline objects. The punning power of ROSP in dominance tests grows as the dimensionality increases.



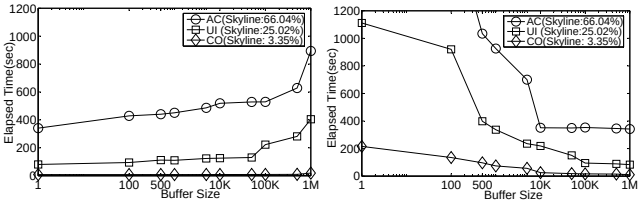
(a) AC (b) UI

Figure 7: Effect of guessing scheme on OSPSSF (n=100K)

5.2.2 The Effect of Partition Buffer Size

Recall that a bounded buffer with the parameter β is used to control partition data loading for local skyline computation in OSPSPF (Algorithm 3) and OSPSOM (Algorithm 5). Figure 8 depicts the effect of different buffer sizes β on both algorithms on 12-dimensional datasets with 1M objects. For OSPSPF, as shown in Figure 8(a), a small buffer size does not greatly impact its performance. With increasing buffer size, the cost increases very slowly

until the buffer size reaches 1M. A steep growth occurs for β=1M, since OSPSPF in this case becomes an instance of OSPSSF, which must sort the entire dataset first. Therefore, we directly set $\beta = 1$ for the main-memory OSPSPF algorithm in all experiments, i.e., the dataset is recursively divided until one skyline object is contained in each partition. On the other hand, the results are different for the secondary memory algorithm OSPSOM, as shown in Figure 8(b). With smaller buffer size (e.g., less than 10,000 objects), the elapsed time increases significantly in all datasets. Especially the running time on the AC dataset ticks to more than 2,000 seconds for a buffer size of one object and to about 1,840 seconds for $\beta = 100$ (these values are out of the plot range). With β larger than 10,000, the costs decrease slowly and become flat. OSPSOM with $\beta = 1M$ is equivalent to an instance of OSPSPF which outputs all skyline objects into the temporary file $\mathcal{T}$. In the remaining experiments, we assume a memory bound of $\beta = 10,000$ for OSPSOM.



(a) β On OSPSPF (b) β On OSPSOM

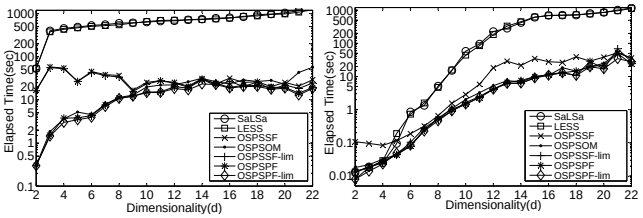
Figure 8: Effect of buffer size (d=12, n=1M)

5.2.3 The Effect of Dimensionality

We study the performance of our proposed methods against SaLSa and LESS, as shown in Figure 9. All reported results are with datasets of 2–22 dimensions and fixed cardinality (n=100K). We do not plot the percentage of skyline points here, since the values are the same as those in Figure 7. The plots show that all our proposed methods greatly outperform SaLSa and LESS in all datasets, but none of them systematically dominates the others. With increasing dimensionality and growth of skyline objects, the execution time of SaLSa and LESS grows at very high levels, however, the cost of our methods grows only slightly. We should mention that for low dimensional UI and CO datasets (e.g., less than 5 dimensions), SaLSa and LESS may outperform some of our proposed methods (such as OSPSSF and OSPSOM) but they are worse than the variants of our methods with the limiting strategy (i.e., OSPSSF-lim and OSPSPF-lim). This is because OSPSSF and OSPSOM require to scan the entire set of ordered objects, but SaLSa can halt earlier and LESS may filter more objects using the EF window. Obviously, OSPSSF-lim and OSPSPF-lim share the ability of SaLSa to stop earlier. Among our proposed skyline methods, OSPSPF-lim comes first, OSPSPF comes second, and then come OSPSSF-lim, OSPSOM, and OSPSSF in order of average performance. Nevertheless, note that OSPSOM assumes a limited memory scenario for our system, whereas the remaining algorithms use unlimited memory to hold the skyline.

5.2.4 The Effect of Data Cardinality

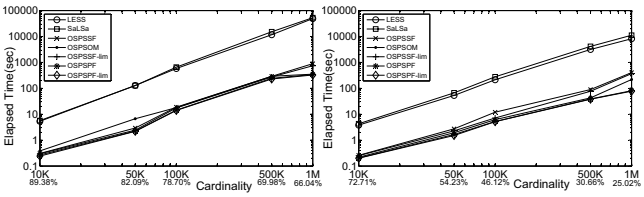
The next experiment evaluates the performance of our methods against LESS and SaLSa with different data sizes (n=10K up to 1M in log scale), as shown in Figure 10. The percentage of skyline objects for each dataset is listed below the x-axis points. The elapsed time of all algorithms increases as data cardinality grows. All our methods are orders of magnitude faster than LESS and SaLSa. For AC datasets, it seems that the performance of OSPSOM depends



(a) AC (b) UI

Figure 9: Effect of dimensionality (n=100K)

on the percentage of skyline points. The higher the percentage of skyline objects out of the entire dataset, the longer time it takes for OSPSOM to terminate. This is because more objects should be swapped out to the temporary file $\mathcal{T}$. From the main-memory algorithms, OSPSPF-lim is the best performing one.



(a) AC (b) UI

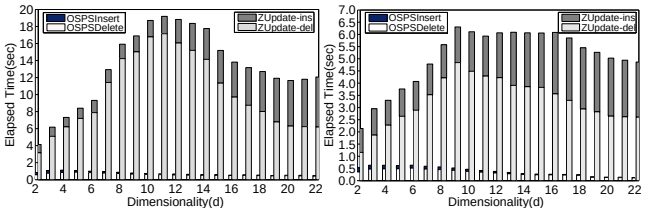
Figure 10: Effect of cardinality(d=12)

5.2.5 Skyline Updates

In this experiment, we compare the cost of updates using our skyline tree versus the update cost on a sophisticated index like the ZBtree [15]. In specific, we compare the cost of applying delete and insert operations to $\mathcal{O}$ while maintaining the skyline in an LCRS tree (we denote these operations by *OSPSDelete* and *OSPSInsert*, respectively) with the cost of the corresponding operations on the maintenance of the ZBtree (denoted by *ZUpdate-del* and *ZUpdate-ins*). Figure 11 illustrates the results. All experiments were conducted on datasets with fixed cardinality (n=1M) for various data dimensionalities (2–22). Compared with deletions, insertions are lightweight operations since they do not need to retrieve all objects dominated by the processed candidates from the source datasets. On the ZBtree, *ZUpdate-del* needs to access all nodes which contain points dominated by the processed candidates and then traverses its skyline ZBtree for dominance tests. Thus, the amount of redundant comparisons among indexing nodes, including the undominated objects in them, weigh on its performance. In addition, all updating operations require expensive skyline ZBtree update costs. However, our methods only need to retrieve all points dominated by the deleted one and then perform dominance tests for them, using our efficient LCRS tree $\mathcal{SL}$. The great pruning power of the LCRS tree $\mathcal{SL}$ in dominance tests can be used for skyline updates, as discussed in Section 3.2 and Section 4.1. This is why the cost of our methods does not increase with dimensionality. On the other hand, the time for ZBtree updates fluctuates.

5.3 Experimental Results on Real Datasets

Our experimental results on the three real datasets are presented in Table 3. Observe that LESS and SaLSa spend much longer time than all our proposed methods for skyline retrieval. For instance, the elapsed times of all our algorithms are bounded by 0.5 seconds, whereas LESS needs about eight seconds and SaLSa requires



(a) AC (b) UI

Figure 11: Skyline updating cost (n=1M)

about ten seconds to complete skyline computation for the NBA dataset. On Household and Color datasets, SaLSa performs better than LESS and both of them are still inferior to all our methods. Consistently to our previous experiments, all of our proposed methods including the variants with limiting strategy work well on these three real datasets. OSPSOM is still the most expensive among our methods and OSPSPF with limiting scheme is the most efficient version of object-based space partitioning skyline algorithms.

Algorithm	NBA	Household	Color
	n=19,181; d=10	n=127,931; d=6	n=68,040; d=9
	skyline% = 57.8	skyline% = 4.51	skyline% = 2.25
LESS	8.207	14.743	0.503
SaLSa	10.120	10.221	0.428
OSPSSF	0.442	0.556	0.116
OSPSSF-lim	0.425	0.538	0.102
OSPSPF	0.432	0.460	0.072
OSPSPF-lim	0.384	0.456	0.068
OSPSOM	0.458	0.783	0.132

Table 3: Elapsed time (sec) on real datasets

6. EXTENSIONS

This section discusses interesting extensions to the object-based space partitioning scheme that we proposed in this paper, demonstrating the versatility of our solutions for computing *k-dominant* skylines and performing *parallel* skyline computation coherently in high dimensional spaces. We also discuss how our methods can be adapted to compute skylines in partially ordered domains.

6.1 k-dominant Skyline Queries using OSP

By relaxing the dominance condition to consider k from the total of d dimensions, the k-dominant skyline query retrieves a representative subset of skyline objects. We say that o k-dominates o', denoted by $o \succ^k o'$, if there is a k sized subset dimensions, such that o dominates o' in the corresponding subspace. [6] show that the transitive property does not hold for the k-dominance relation; i.e., there could be $o \succ^k o'$, $o' \succ^k o''$, and $o \not\succ^k o''$. The reason for this is that different k dimensional subsets could be used for $o \succ^k o'$ and $o' \succ^k o''$. This adds to the complexity of k-dominant skyline computation. However, from [6], we know that for an object $o \in \mathcal{O}$ to be in the k-dominance skyline, o should also be part of the conventional skyline considering all dimensions. Thus, it is sufficient to detect whether a candidate is in the k-dominate skyline using our LCRS tree $\mathcal{SL}$ instead of the entire dataset $\mathcal{O}$. We now highlight the properties, based on the definition of k-dominance, which facilitate k-dominance tests using the LCRS tree $\mathcal{SL}$.

- For any node $e \in \mathcal{SL}$, its reference object $e.o$ must k-dominate all objects in its child nodes whose partition address contains at least k 1-bits
- For any node $e \in \mathcal{SL}$, all objects in its child partitions with address containing less than $(d-k)$ 1-bits must k-dominate its reference object $e.o$
- For any node $e \in \mathcal{SL}$, any object $o \in \mathcal{O}$ with address $A_{e.o}(o)$ containing at least (k) 1-bits must be k-dominated by its reference object $e.o$
- For any node $e \in \mathcal{SL}$, any object $o \in \mathcal{O}$ with address $A_{e.o}(o)$ containing less than $(d-k)$ 1-bits must k-dominate its reference object $e.o$

We can design an algorithm that computes k-dominant skylines, by traversing the tree and pruning skyline objects using the above properties. More specifically, for each encountered node, we directly apply the pruning rules to eliminate nodes in $\mathcal{SL}$ that are k-dominated.

6.2 Parallel Skyline Evaluation using OSP

The conventional technique for parallel skyline evaluation is to partition the entire dataset to several servers, compute the local skylines in each partition and then merge them. Intuitively, a careful workload assignment strategy can reduce the size of local skylines and improve the efficiency of their merging. Our object-based space partitioning scheme is a natural method for this purpose, as it implies the dominance relations among partitions and the dominance tests among incomparable partitions can be omitted effectively. A straightforward strategy is to divide the dataset with respect to an object and then group all incomparable partitions to balance the workload at different servers. At the same time, a LCRS partition tree among all distributed servers is constructed to avoid the communications between incomparable servers. Another advantage of workload assignment based on the object-based space partitioning scheme is that it can handle arbitrary data distributions by dynamically splitting the data space. We note that here we only sketch a method for deriving an appropriate partitioning scheme for parallel computation, in the same spirit as the space-partitioning approach proposed in [29]. Specific optimizations are out of the scope of this paper and they are subject for future work.

6.3 Skyline Evaluation on POD using OSP

So far, we have assumed that the domains of all dimensions define a total order. In practice, we may have dimensions with partially ordered domains, where the values of two objects can be incomparable. For example, consider a categorical dimension with four values {grey, red, green, white}, such that grey is preferable to red or green, red or green is preferable to white, but there is no clear preference between red and green. In this case, a partially ordered domain (POD) {(grey), (red, green), (white)} is defined for this dimension. If two objects have incomparable values in a dimension with POD, then by definition they are incomparable (i.e., one does not dominate the other).

In this section, we discuss how our OSP indexing method can be extended for data with dimensions having POD. Basically, we employ the same OSP scheme described in Section 3.1, but this time we also use the partition with address $00\ldots0$,[4] where we store objects which are incomparable to the reference object o in the PODs. We call this the *Partially Incomparable Partition* ($\mathcal{PIP}$) w.r.t. the reference o.

For each object $o' \in \mathcal{PIP}$ w.r.t. o, we extend Definition 1, to define its **partial partition address** $A_o(o')$: for each dimension i, if $o_i < o'_i$ or o_i and o'_i are incomparable, then $A_o(o')[i]=1$;

[4]Recall that this partition is normally disregarded; it is empty since the reference object is a skyline object.

otherwise, $A_o(o')[i]$=0. Accordingly, the objects in $\mathcal{PIP}$ are partitioned into **partial locating partitions** based on their partial partition address. The dominating and dominated partition sets of an object o' (i.e., $\mathrm{D}_o(o')$ and $\mathrm{U}_o(o')$, respectively) are then extended to include the dominating and dominated partial locating partitions in the $\mathcal{PIP}$. In other words, the object $o' \in \mathcal{PIP}$ w.r.t. o may be dominated by (resp. dominate) some objects in the dominating (resp. dominated) partitions of its partial locating partition.

Therefore, during PreOderDominate, each object $o' \in \mathcal{PIP}$ w.r.t. the reference object $e.o$ of a LCRS skyline tree node e, should do dominance tests with objects in the partitions of $\mathrm{D}_o(o')$ and it could finally be inserted into $\mathcal{PIP}$. $\mathcal{PIP}$ could dynamically be re-partitioned, like normal partitions. In the adapted version of OSPSOnPartitioningFirst for PODs, the local LCRS skyline tree $\mathcal{SL}$ for $\mathcal{PIP}$ is created first, but the filtering process for $\mathcal{PIP}$ is different compared to regular partitions, since some objects in $\mathcal{SL}$ may be dominated by objects in its sibling partitions that follow. Due to this, nodes may have to be deleted in the $\mathcal{SL}$ constructed during the progress of the algorithm. For deleted nodes, we can use the trick mentioned in Section 4.1 (i.e., mark them as "non-skyline", instead of explicitly deleting them to avoid expensive re-organization of the tree). By applying these changes to the algorithms, we can use them if some dimensions have partially ordered domains.

7. CONCLUSIONS

In this paper, we proposed an efficient set of skyline evaluation algorithms that are based on the idea of organizing the discovered skyline points in a tree which defines a recursive space partitioning. With the help of this tree, each candidate skyline object only needs to be compared for dominance with a small subset of the existing skyline points. The nice feature of this technique is that the ratio of skyline points to be compared with a candidate decreases with the dimensionality of the problem. This makes our solutions scalable to the dimensionality, a feature that all previously proposed skyline algorithms lack.

By accessing the data in a particular (topological-sort) order, we guarantee that each object that is inserted to the tree is a skyline object and cannot be pruned by objects accessed later. This ensures efficient updates into the tree, as data reorganization is avoided. Our first algorithm directly applies this idea after having pre-sorted the data. Our second method partitions the dataset recursively, while constructing the tree and achieves better performance because it avoids sorting. Finally, we propose a version of the partitioning-based algorithm, which is appropriate for the case where the skyline is larger than the available memory.

Our experimental results confirm that dominance checks dominate the cost of skyline computation and show that our methods are orders of magnitude faster than the state-of-the-art, for problems of high-dimensionality and anti-correlated data. In the future, we plan to identify and compare more appropriate heuristics for selecting the skyline objects that define the space partitioning, as the choice of these objects has significant effect in the performance.

8. REFERENCES

[1] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *ACM TODS*, 33(4):1–45, 2008.

[2] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. In *SODA*, 1990.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[4] C. Y. Chan, P. K. Eng, and K. L. Tan. Efficient processing of skyline queries with partially-ordered domains. In *ICDE*, 2005.

[5] C. Y. Chan, P. K. Eng, and K. L. Tan. Stratified computation of skylines with partially-ordered domains. In *SIGMOD*, 2005.

[6] 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*, 2006.

[7] C. Y. Chan, H. V. Jagadish, K. L. Tan, A. K. H. Tung, and Z. Zhang. On high dimensional skylines. In *EDBT*, 2006.

[8] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *ICDE*, 2006.

[9] L. Chen and X. Lian. Dynamic skyline queries in metric spaces. In *EDBT*, 2008.

[10] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, 2003.

[11] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, 2005.

[12] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. Skyline query processing for incomplete data. In *ICDE*, 2008.

[13] W. Kießling. Foundations of preferences in database systems. In *VLDB*, 2002.

[14] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: an online algorithm for skyline queries. In *VLDB*, 2002.

[15] K. C. K. Lee, B. Zheng, H. Li, and W. C. Lee. Approaching the skyline in z order. In *VLDB*, 2007.

[16] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *SIGMOD*, 2008.

[17] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *ICDE*, 2007.

[18] M. Morse, J. M. Patel, and H. V. Jagadish. Efficient skyline computation over low-cardinality domains. In *VLDB*, 2007.

[19] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM TODS*, 30(1):41–82, 2005.

[20] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *VLDB*, 2007.

[21] D. Sacharidis, P. Bouros, and T. K. Sellis. Caching dynamic skyline queries. In *SSDBM*, 2008.

[22] N. Sarkas, G. Das, N. Koudas, and A. K. H. Tung. Categorical skylines for streaming data. In *SIGMOD*, 2008.

[23] K. L. Tan, P. K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, 2001.

[24] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *ICDE*, 2006.

[25] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *SIGMOD*, 2008.

[26] S. Wang, Q. H. Vu, B. C. Ooi, A. K. H. Tung, and L. Xu. Skyframe: a framework for skyline query processing in peer-to-peer systems. *VLDB J.*, 18, 2009.

[27] R. C. W. Wong, A. W. Fu, J. Pei, Y. S. Ho, T. Wong, and Y. Liu. Efficient skyline querying with variable user preferences on nominal attributes. In *VLDB*, 2008.

[28] R. C. W. Wong, J. Pei, A. W. C. Fu, and K. Wang. Mining favorable facets. In *KDD*, 2007.

[29] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *EDBT*, 2006.

[30] M. L. Yiu and N. Mamoulis. Efficient processing of top-k dominating queries on multi-dimensional data. In *VLDB*, 2007.

[31] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, 2005.