Noname manuscript No.
(will be inserted by the editor)

Multi-Dimensional Top-k Dominating Queries

Man Lung Yiu · Nikos Mamoulis

Abstract The top-k dominating query returns k data objects which dominate the highest number of objects in a dataset. This query is an important tool for decision support since it provides data analysts an intuitive way for finding significant objects. In addition, it combines the advantages of top-k and skyline queries without sharing their disadvantages: (i) the output size can be controlled, (ii) no ranking functions need to be specified by users, and (iii) the result is independent of the scales at different dimensions. Despite their importance, top-k dominating queries have not received adequate attention from the research community. This paper is an extensive study on the evaluation of top-k dominating queries. First, we propose a set of algorithms that apply on indexed multi-dimensional data. Second, we investigate query evaluation on data that are not indexed. Finally, we study a relaxed variant of the query which considers dominance in dimensional subspaces. Experiments using synthetic and real datasets demonstrate that our algorithms significantly outperform a previous skyline-based approach. We also illustrate the applicability of this multi-dimensional analysis query by studying the meaningfulness of its results on real data.

Keywords: Top-k Retrieval, Preference Dominance, Score Counting

1 Introduction

Consider a dataset $\mathcal{D}$ of points in a d-dimensional space $\mathcal{R}^d$. Given a (monotone) ranking function $F : \mathcal{R}^d \rightarrow \mathcal{R}$, a *top-$k$ query* [14,9] returns k points with the smallest F value. For example, Figure 1a shows a set of hotels modeled by points in the 2D space, where the dimensions correspond to (preference) attribute values; traveling time to a conference venue and room price. For the ranking function $F = x+y$, the top-2 hotels are p_4 and p_6. An obvious advantage of the top-k query is that the user is able to control the number of results (through the parameter k). On the other hand, it might not always be easy for the user to specify an appropriate ranking function. In addition, there is no straightforward way for a data analyst to identify the most important objects using top-k queries, since different functions may infer different rankings.

A *skyline query* [2] retrieves all points which are not dominated by any other point. Assuming that smaller values are preferable to larger at all dimensions, a point p *dominates* another point p' (i.e., $p \succ p'$) when

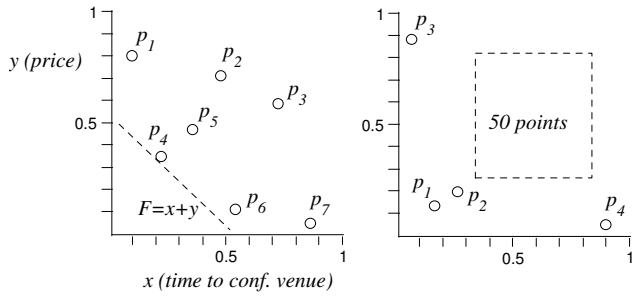
$$(\exists\ i \in [1,d],\ p[i] < p'[i]) \wedge (\forall\ i \in [1,d],\ p[i] \leq p'[i])$$

where $p[i]$ denotes the coordinate of p in the i-th dimension. Continuing with the example in Figure 1a, the skyline query returns points p_1, p_4, p_6, and p_7. [2] showed that the skyline contains the top-1 result for any monotone ranking function; therefore, it can be used by decision makers to identify potentially important objects to some database users. A key advantage of the skyline query is that it does not require the use of a specific ranking function; its results only depend on the intrinsic characteristics of the data. Furthermore, the skyline is not affected by potentially different scales at different dimensions (monetary unit or time unit in the example of Figure 1a); only the order of the dimensional projections of the objects is important. On the other hand, the size of the skyline cannot be controlled by the user and it can be as large as the data size in the worst case. As a result, the user may be overwhelmed as she may have to examine numerous skyline points manually in order to identify the ones that will eventually be regarded as important. In fact, the skyline

Man Lung Yiu
Department of Computer Science, Aalborg University
E-mail: mly@cs.aau.dk

Nikos Mamoulis
Department of Computer Science, University of Hong Kong
E-mail: nikos@cs.hku.hk

may not be used as an informative and concise summary for the dataset. It is well known that [2]: for a fully correlated dataset, the skyline contains exactly 1 point, which is not informative about the distribution of other data points; for a totally anti-correlated dataset, the skyline is the whole dataset, which is definitely not a concise data summary.



(a) dataset with 7 hotels (b) dataset with 54 hotels

Fig. 1 Features of hotels

To summarize, top-k queries do not provide an objective order of importance for the points, because their results are sensitive to the preference function used. Skyline queries, on the other hand, only provide a subset of important points, which may have arbitrary size. To facilitate analysts, who may be interested in a natural order of importance, according to dominance, we propose the following intuitive score function:

$$\tau(p) = |\ \{\ p' \in \mathcal{D} \mid p \succ p'\ \}\ | \tag{1}$$

In words, the *score* $\tau(p)$ is the number of points dominated by point p. The following monotone property holds for τ:

$$\forall\ p, p' \in \mathcal{D},\ p \succ p' \Rightarrow \tau(p) > \tau(p') \tag{2}$$

Based on the τ function, we can define a natural ordering of the points in the database. Accordingly, the *top-k dominating* query returns k points in $\mathcal{D}$ with the highest score. For example, the top-2 dominating query on the data of Figure 1a retrieves p_4 (with $\tau(p_4) = 3$) and p_5 (with $\tau(p_5) = 2$). This result may indicate to a data analyst (i.e., conference organizer) the most popular hotels to the conference participants (considering price and traveling time as selection factors). Here the popularity of a hotel p is defined based on over how many other hotels would p be preferred, for any preference function.

As another example on how the τ function is related to popularity, consider a dataset with 54 hotels, as shown in Figure 1b. 50 of these points are not shown explicitly; the figure only illustrates a rectangle which includes all of them. The top-2 dominating points in this case are p_1 (with $\tau(p_1) = 51$) and p_2 (with $\tau(p_2) = 50$). Even though p_2 is not a skyline point, it becomes important after the top-1 hotel p_1 has been fully booked. The reason is that p_2 is guaranteed to be better than at least 50 points, regardless of any monotone preference ranking function considered by individual conference participants. On the other hand, skyline point p_3 may not provide such guarantee; in the worst case, all conference participants may just be looking for cheap hotels, so p_3 is no good at all. A similar observation holds for the skyline point p_4.

The above examples illustrate that a top-k dominating query is a powerful decision support tool, since it identifies the most significant objects in an intuitive way. From a practical perspective, top-k dominating queries combine the advantages of top-k queries and skyline queries without sharing their disadvantages. The number of results can be controlled without specifying any ranking function. In addition, data normalization is not required; the results are not affected by different scales or data distributions at different dimensions.

The top-k dominating query was first introduced by Papadias et al. [24] as an extension of the skyline query. However, the importance and practicability of the query was not identified there. This paper is an extensive study of this analysis query. We note that the R-tree (used in [24]) may not be the most appropriate index for this query; since computing $\tau(p)$ is in fact an *aggregate* query, we can replace the R-tree by an *aggregate R-tree* (aR-tree) [17,23]. In addition, we observe that the skyline-based approach proposed in [24] may perform many unnecessary score countings, since the skyline could be much larger than k.

Motivated by these observations, our first contribution includes two specialized and very efficient methods for evaluating top-k dominating queries on a dataset indexed by an aR-tree. We propose (i) a batch counting technique for computing scores of multiple points simultaneously, (ii) a counting-guided search algorithm for processing top-k dominating queries, and (iii) a priority-based tree traversal algorithm that retrieves query results by examining each tree node at most once. We enhance the performance of (ii) with *lightweight counting*, which derives relatively tight upper bound scores for non-leaf tree entries at low I/O cost. Furthermore, to our surprise, the intuitive *best-first* traversal order [13,24] turns out not to be the most efficient for (iii) because of potential partial dominance relationships between visited entries. Thus, we perform a careful analysis on (iii) and propose a *novel, efficient tree traversal order* for it. Extensive experiments show that our methods significantly outperform the skyline-based approach of [24].

The above algorithms have been published in the preliminary version of this paper [31], where we also propose top-k dominating query variants such as *aggregate* top-k dominating queries and *bichromatic* top-k dominating queries; these extensions are not further investigated here. Instead, in this paper, we examine two alternative topics relevant to top-k dominating queries. The first is the processing of top-k dominating queries on non-indexed data. In certain scenarios

(e.g., dynamically generated data), it is not always reasonable to assume an existing aR-tree index for them a priori. In view of this, we propose a method that evaluates top-k dominating queries by accessing the (unordered) data only a few times. As we demonstrate experimentally, this method significantly outperforms the best index-based method, which requires the bulk-loading an aR-tree index before evaluation. Our second extension over [31] is the proposal and study of a relaxed form of the top-k dominating query. In this query variant, the $\tau(p)$ score is defined by the number of dimensional subspaces where point p dominates another point p'. As we demonstrate, this query derives more meaningful results than the basic top-k dominating query.

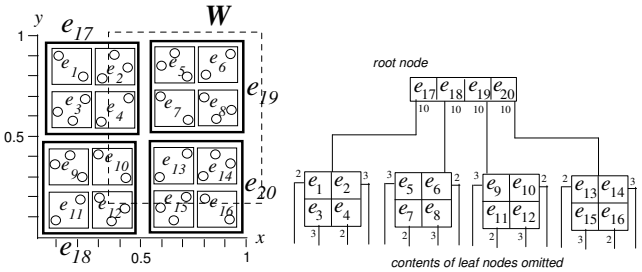
The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 discusses the properties of top-k dominating search and proposes optimizations for the existing solution in [24]. We then propose eager/lazy approaches for evaluating top-k dominating queries. Section 4 presents an eager approach that guides the search by deriving tight score bounds for encountered non-leaf tree entries immediately. Section 5 develops an alternative, lazy approach that defers score computation of visited entries and gradually refines their score bounds when more tree nodes are accessed. Section 6 presents techniques for processing top-k dominating queries on non-indexed data. Section 7 introduces the relaxed top-k dominating query and discusses its evaluation. In Section 8, experiments are conducted on both real and synthetic datasets to demonstrate that the proposed algorithms are efficient and also top-k dominating queries return meaningful results to users. Section 9 summarizes our experimental findings and discusses the case of high dimensional data. Finally, Section 10 concludes the paper.

2 Related Work

Top-k dominating queries include a counting component, i.e., multi-dimensional aggregation; thus, we review related work on spatial aggregation processing. In addition, as the dominance relationship is relevant to skyline queries, we survey existing methods for computing skylines.

2.1 Spatial Aggregation Processing

R-trees [12] have been extensively used as access methods for multi-dimensional data and for processing spatial queries, e.g., range queries, nearest neighbors [13], and skyline queries [24]. The aggregate R-tree (aR-tree) [17,23] augments to each non-leaf entry of the R-tree an aggregate measure of all data points in the subtree pointed by it. It has been used to speed up the evaluation of spatial aggregate queries, where measures (e.g., number of buildings) in a spatial region (e.g., a district) are aggregated.



(a) a set of points (b) a COUNT aR-tree

Fig. 2 aR-tree example

Figure 2a shows a set of points in the 2D space, indexed by the COUNT aR-tree in Figure 2b. Each non-leaf entry stores the COUNT of data points in its subtree. For instance, in Figure 2b, entry e_{17} has a count 10, meaning that the subtree of e_{17} contains 10 points. Suppose that a user asks for the number of points intersecting the region W, shown in Figure 2a. To process the query, we first examine entries in the root node of the tree. Entries that do not intersect W are pruned because their subtree cannot contain any points in W. If an entry is spatially covered by W (e.g., entry e_{19}), its count (i.e., 10) is added to the answer without accessing the corresponding subtree. Finally, if a non-leaf entry intersects W but it is not contained in W (e.g., e_{17}), search is recursively applied to the child node pointed by the entry, since the corresponding subtree may contain points inside or outside W. Note that the counts augmented in the entries effectively reduce the number of accessed nodes. To evaluate the above example query, only 10 nodes in the COUNT aR-tree are accessed but 17 nodes in an R-tree with the same node capacity would be visited.

2.2 Skyline Computation

Börzsönyi et al. [2] were the first to propose efficient external memory algorithms for processing skyline queries. The BNL (block-nested-loop) algorithm scans the dataset while employing a bounded buffer for tracking the points that cannot be dominated by other points in the buffer. A point is reported as a result if it cannot be dominated by any other point in the dataset. On the other hand, the DC (divide-and-conquer) algorithm recursively partitions the dataset until each partition is small enough to fit in memory. After the local skyline in each partition is computed, they are merged to form the global skyline. The BNL algorithm was later improved to SFS (sort-filter-skyline) [8] and LESS (linear elimination sort for skyline) [11] in order to optimize the average-case running time.

The above algorithms are generic and applicable for non-indexed data. On the other hand, [28,16,24] exploit data in-

dexes to accelerate skyline computation. The state-of-the-art algorithm is the BBS (branch-and-bound skyline) algorithm [24], which is shown to be I/O optimal for computing skylines on datasets indexed by R-trees.

Recently, the research focus has been shifted to the study of queries based on variants of the dominance relationship. [22] aims at extracting from the skyline points a k-sized subset such that it dominates the maximum number of data points; in other words, the result set cannot contain any non-skyline point. [20] proposes a data cube structure for speeding up the evaluation of queries that analyze the dominance relationship of points in the dataset. However, incremental maintenance of the data cube over updates has not been addressed in [20]. Clearly, it is prohibitively expensive to recompute the data cube from scratch for dynamic datasets with frequent updates. [6] identifies the problem of computing *top-k frequent skyline* points, where the frequency of a point is defined by the number of dimensional subspaces. [5] studies the *k-dominant skyline* query, which is based on the k-dominance relationship. A point p is said to k-dominate another point p' if p dominates p' in at least one k-dimensional subspace. The k-dominant skyline contains the points that are not k-dominated by any other point. When k decreases, the size of the k-dominant skyline also decreases. Observe that [22, 20, 6, 5] cannot be directly applied to evaluate top-k dominating queries studied in this paper.

Finally, [32, 26, 27, 25] study the efficient computation of skylines for every subspace; [29] proposes a technique for retrieving the skyline for a given subspace; [1, 15] investigate skyline computation over distributed data; [10, 7] develop techniques for estimating the skyline cardinality; [21] studies continuous maintenance of the skyline over a data stream; and [4] addresses skyline computation over datasets with partially-ordered attributes.

3 Preliminary

In this section, we discuss some fundamental properties of top-k dominating search, assuming that the data have been indexed by an aR-tree. In addition, we propose an optimized version for the existing top-k dominating algorithm [24] that operates on aR-trees.

3.1 Score Bounding Functions

Before presenting our top-k dominating algorithms, we first introduce some notation that will be used in this paper. For an aR-tree entry e (i.e., a minimum bounding box) whose projection on the i-th dimension is the interval $[e[i]^-, e[i]^+]$, we denote its lower corner e^- and upper corner e^+ by

$$e^- = (e[1]^-, e[2]^-, \cdots, e[d]^-)$$

$$e^+ = (e[1]^+, e[2]^+, \cdots, e[d]^+)$$

Observe that both e^- and e^+ do not correspond to actual data points but they allow us to express dominance relationships among points and minimum bounding boxes conveniently. As Figure 3 illustrates, there are three cases for a point to dominate a non-leaf entry. Since $p_1 \succ e_1^-$ (i.e., full dominance), p_1 must also dominate *all* data points indexed under e_1. On the other hand, point p_2 dominates e_1^+ but not e_1^- (i.e., partial dominance), thus p_2 dominates some, but not all data points in e_1. Finally, as $p_3 \not\succ e_1^+$ (i.e., no dominance), p_3 cannot dominate any point in e_1. Similarly, the cases for an entry to dominate another entry are: (i) full dominance (e.g., $e_1^+ \succ e_3^-$), (ii) partial dominance (e.g., $e_1^- \succ e_4^+ \wedge e_1^+ \not\succ e_4^-$), (iii) no dominance (e.g., $e_1^- \not\succ e_2^+$).

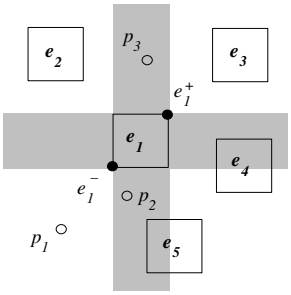


Fig. 3 Dominance relationship among aR-tree entries

Given a tree entry e, whose sub-tree has not been visited, $\tau(e^+)$ and $\tau(e^-)$ correspond to the *tightmost* lower and upper score bounds respectively, for any point indexed under e. As we will show later, $\tau(e^+)$ and $\tau(e^-)$ can be computed by a search procedure that accesses only aR-tree nodes that intersect e along at least one dimension. These bounds help pruning the search space and defining a good order for visiting aR-tree nodes. Later in Sections 4 and 5, we replace the tight bounds $\tau(e^+)$ and $\tau(e^-)$ with loose lower and upper bounds for them ($\tau^l(e)$ and $\tau^u(e)$, respectively). Bounds $\tau^l(e)$ and $\tau^u(e)$ are cheaper to compute and can be progressively refined during search, therefore trading-off between computation cost and bound tightness. The computation and use of score bounds in practice will be further elaborated there.

3.2 Optimizing the Skyline-Based Approach

Papadias et al. [24] proposed a Skyline-Based Top-k Dominating Algorithm (STD) for top-k dominating queries, on data indexed by an R-tree. They noted that the skyline is guaranteed to contain the top-1 dominating point, since a non-skyline point has lower score than at least one skyline point that dominates it (see Equation 2). Thus, STD retrieves the skyline points, computes their τ scores and outputs the point p with the highest score. It then removes p from the

dataset, incrementally finds the skyline of the remaining points, and repeats the same process.

Consider for example a top-2 dominating query on the dataset shown in Figure 4. STD first retrieves the skyline points p_1, p_2, and p_3 (using the BBS skyline algorithm of [24]). For each skyline point, a range query is issued to count the number of points it dominates. After that, we have $\tau(p_1) = 1$, $\tau(p_2) = 4$, and $\tau(p_3) = 1$. Hence, p_2 is reported as the top-1 result. We now restrict the region of searching for the next result. First, Equation 2 suggests that the region dominated by the remaining skyline points (i.e., p_1 and p_3) needs not be examined. Second, the region dominated by p_2 (i.e., the previous result) may contain some points which are not dominated by the remaining skyline points p_1 and p_3. It suffices to retrieve the skyline points (i.e., p_4 and p_5) in the constrained (gray) region M shown in Figure 4. After counting their scores using the tree, we have $\tau(p_4) = 2$ and $\tau(p_5) = 1$. Finally, we compare them with the scores of retrieved points (i.e., p_1 and p_3) and report p_4 as the next result.

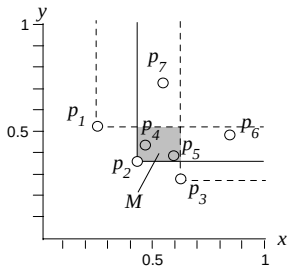


Fig. 4 Constrained skyline

In this section, we present two optimizations that greatly reduce the I/O cost of the above solution by exploiting aR-trees. Our first optimization is called *batch counting*. Instead of iteratively applying separate range queries to compute the scores of the skyline points, we perform them in batch. Algorithm 1 shows the pseudo-code of this recursive batch counting procedure. It takes two parameters: the current aR-tree node Z and the set of points V, whose τ scores are to be counted. Initially, Z is set to the root node of the tree and $\tau(p)$ is set to 0 for each $p \in V$. Let e be the current entry in Z to be examined. As illustrated in Section 3.1, if e is a non-leaf entry and there exists some point $p \in V$ such that $p \succ e^+ \wedge p \not\succ e^-$, then p may dominate some (but not guaranteed to dominate all) points indexed under e. Thus, we cannot immediately decide the number of points in e dominated by p. In this case, we have to invoke the algorithm recursively on the child node pointed by e. Otherwise, for each point $p \in V$, its score is incremented by $\texttt{COUNT}(e)$ when it dominates e^-. BatchCount correctly computes the τ score for all $p \in V$, at a single tree traversal.

Algorithm 2 is a pseudo-code of the Iterative Top-k Dominating Algorithm (ITD), which optimizes the STD algorithm of [24]. Like STD, ITD computes the top-k dominating points iteratively. In the first iteration, ITD computes in V' the skyline of the whole dataset, while in subsequent iterations, the computation is *constrained* to a region M. M is the region dominated by the reported point q in the previous iteration, but not any point in the set V of retrieved points in past iterations. At each loop, Lines 6–8 compute the scores for the points in V' in batches of B points each ($B \leq |V'|$). By default, the value of B is set to the number of points that can fit into a memory page. Our second optimization is that we sort the points in V' by a space-filling curve (Hilbert ordering) [3] before applying batch counting, in order to increase the compactness of the MBR of a batch. After merging the constrained skyline with the global one, the object q with the highest τ score is reported as the next dominating object, removed from V and used to compute the constrained skyline at the next iteration. The algorithm terminates after k objects have been reported.

Algorithm 1 Batch Counting

```
   algorithm BatchCount(Node Z, Point set V)
1: for all entries e ∈ Z do
2:     if Z is non-leaf and ∃p ∈ V, p ≻ e+ ∧ p ⊁ e− then
3:         read the child node Z′ pointed by e;
4:         BatchCount(Z′, V);
5:     else
6:         for all points p ∈ V do
7:             if p ≻ e− then
8:                 τ(p):=τ(p)+COUNT(e);
```

For instance, in Figure 4, q corresponds to point $(0, 0)$ and $V = \varnothing$ in the first loop, thus M corresponds to the whole space and the whole skyline $\{p_1, p_2, p_3\}$ is stored in V', the points there are sorted and split in batches and their τ scores are counted using the BatchCount algorithm. In the beginning of the second loop, $q = p_2$, $V = \{p_1, p_3\}$, and M is the gray region in the figure. V' now becomes $\{p_4, p_5\}$ and the corresponding scores are batch-counted. The next point is then reported (e.g., p_4) and the algorithm continues as long as more results are required.

Algorithm 2 Iterative Top-k Dominating Algorithm (ITD)

```
   algorithm ITD(Tree R, Integer k)
 1: V:=∅; q:=origin point;
 2: for i := 1 to k do
 3:     M:=region dominated by q but by no point in V;
 4:     V′:=skyline points in M;
 5:     sort the points in V′ by Hilbert ordering;
 6:     for all batches V_c of (B) points in V′ do
 7:         initialize all scores of points in V_c to 0;
 8:         BatchCount(R.root,V_c);
 9:     V:=V ∪ V′;
10:     q:=the point with maximum score in V;
11:     remove q from V;
12:     report q as the i-th result;
```

4 Counting-Guided Search

The skyline-based solution becomes inefficient for datasets with large skylines as τ scores of many points are computed. In addition, not all skyline points have large τ scores. Motivated by these observations, we study algorithms that solve the problem directly, without depending on skyline computations. This section presents an *eager* approach for the evaluation of top-k dominating queries, which traverses the aR-tree and computes tight upper score bounds for encountered non-leaf tree entries immediately; these bounds determine the visiting order for the tree nodes. We discuss the basic algorithm, develop optimizations for it, and investigate by an analytical study the improvements of these optimizations.

4.1 The Basic Algorithm

Recall from Section 3.1 that the score of any point p indexed under an entry e is upper-bounded by $\tau(e^-)$. Based on this observation, we can design a method that traverses aR-tree nodes in descending order of their (upper bound) scores. The rationale is that points with high scores can be retrieved early and accesses to aR-tree nodes that do not contribute to the result can be avoided.

Algorithm 3 shows the pseudo code of the Simple Counting-Guided Algorithm (SCG), which directs search by counting upper bound scores of examined non-leaf entries. A max-heap H is employed for organizing the entries to be visited in descending order of their scores. W is a min-heap for managing the top-k dominating points as the algorithm progresses, while γ is the k-th score in W (used for pruning). First, the upper bound scores $\tau(e^-)$ of the aR-tree root entries are computed in batch (using the BatchCount algorithm) and these are inserted into the max-heap H. While the score $\tau(e^-)$ of H's top entry e is higher than γ (implying that points with scores higher than γ may be indexed under e), the top entry is deheaped, and the node Z pointed by e is visited. If Z is a non-leaf node, its entries are enheaped, after BatchCount is called to compute their upper score bounds. If Z is a leaf node, the scores of the points in it are computed in batch and the top-k set W (also γ) is updated, if applicable.

As an example, consider the top-1 dominating query on the set of points in Figure 5. There are 3 leaf nodes and their corresponding entries in the root node are e_1, e_2, and e_3. First, upper bound scores for the root entries (i.e., $\tau(e_1^-) = 3$, $\tau(e_2^-) = 7$, $\tau(e_3^-) = 3$) are computed by the batch counting algorithm, which incurs 3 node accesses (i.e., the root node and leaf nodes pointed by e_1 and e_3). Since e_2 has the highest upper bound score, the leaf node pointed by e_2 will be accessed next. Scores of entries in e_2 are computed in batch and we obtain $\tau(p_1) = 5$, $\tau(p_2) = 1$, $\tau(p_3) = 2$. Since p_1 is a point and $\tau(p_1)$ is higher than the scores of remaining entries (p_2, p_3, e_1, e_3), p_1 is guaranteed to be the top-1 result.

Algorithm 3 Simple Counting Guided Algorithm (SCG)

```
   algorithm SCG(Tree R, Integer k)
 1: H:=new max-heap; W:=new min-heap;
 2: γ:=0;                                  ▷ the k-th highest score found so far
 3: BatchCount(R.root,{e⁻ | e ∈ R.root});
 4: for all entries e ∈ R.root do
 5:     enheap(H, ⟨e, τ(e⁻)⟩);
 6: while |H| > 0 and H's top entry's score > γ do
 7:     e:=deheap(H);
 8:     read the child node Z pointed by e;
 9:     if Z is non-leaf then
10:         BatchCount(R.root,{e_c⁻ | e_c ∈ Z});
11:         for all entries e_c ∈ Z do
12:             enheap(H, ⟨e_c, τ(e_c⁻)⟩);
13:     else                                  ▷ Z is a leaf
14:         BatchCount(R.root,{p | p ∈ Z});
15:         update W and γ, using ⟨p, τ(p)⟩, ∀p ∈ Z
16: report W as the result;
```

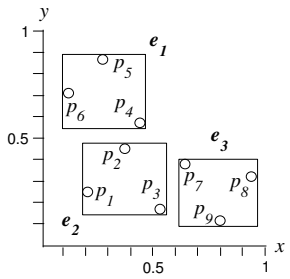


Fig. 5 Computing upper bound scores

4.2 Optimizations

Now, we discuss three optimizations that can greatly reduce the cost of the basic SCG. First, we utilize encountered data points to strengthen the pruning power of the algorithm. Next, we apply a lazy counting method that delays the counting for points, in order to form better groups for batch counting. Finally, we develop a lightweight technique for deriving upper score bounds of non-leaf entries at low cost.

The pruner set. SCG visits nodes and counts the scores of points and entries, based only on the condition that the upper bound score of their parent entry is greater than γ. However, we observe that points which have been counted, but have scores at most γ can also be used to prune early other entries or points, which are dominated by them.[1] Thus, we maintain a pruner set F, which contains points that (i) have been counted exactly (i.e., at Line 15), (ii) have scores at most γ, and (iii) are not dominated by any other point in F. The third condition ensures that only minimal information

[1] Suppose that a point p satisfies $\tau(p) \leq \gamma$. Applying Equation 2, if a point p' is dominated by p, then we have $\tau(p') < \gamma$.

is kept in F.[2] We perform the following changes to SCG in order to use F. First, after deheaping an entry e (Line 7), we check whether there exists a point $p \in F$, such that $p \succ e^-$. If yes, then e is pruned and the algorithm goes back to Line 6. Second, before applying BatchCount at Lines 10 and 14, we eliminate any entries or points that are dominated by a point in F.

Lazy counting. The performance of SCG is negatively affected by executions of BatchCount for a small number of points. A batch may have few points if many points in a leaf node are pruned with the help of F. In order to avoid this problem, we employ a *lazy counting* technique, which works as follows. When a leaf node is visited (Line 13), instead of directly performing batch counting for the points p, those that are not pruned by F are inserted into a set L, with their upper bound score $\tau(e^-)$ from the parent entry. If, after an insertion, the size of L exceeds B (the size of a batch), then BatchCount is executed for the contents of L, and all W, γ, F are updated. Just before reporting the final result set (Line 16), batch counting is performed for potential results $p \in L$ not dominated by any point in F and with upper bound score greater than γ. We found that the combined effect of the pruner set and lazy counting lead to 30% I/O cost reduction of SCG, in practice.

Lightweight upper bound computation. As mentioned in Section 3.1, the tight upper score bound $\tau(e^-)$ can be replaced by a looser, cheaper to compute, bound $\tau^u(e)$. We propose an optimized version of SCG, called Lightweight Counting Guided Algorithm (LCG). Line 10 of SCG (Algorithm 3) is replaced by a call to LightBatchCount, which is a variation of BatchCount. In specific, when bounds for a set V of *non-leaf* entries are counted, the algorithm avoids expensive accesses at aR-tree leaf nodes, but uses entries at non-leaf nodes to derive looser bounds.

LightBatchCount is identical to Algorithm 1, except that the recursion of Line 2 is applied when Z is at least two levels above leaf nodes and there is a point in V that partially dominates e; thus, the else statement at Line 5 now refers to nodes one level above the leaves. In addition, the condition at Line 7 is replaced by $p \succ e^+$; i.e., $\texttt{COUNT}(e)$ is added to $\tau^u(p)$, even if p partially dominates entry e.

As an example, consider the three root entries of Figure 5. We can compute loose upper score bounds for $V = \{e_1^-, e_2^-, e_3^-\}$, without accessing the leaf nodes. Since, e_2^- fully dominates e_2 and partially dominates e_1, e_3, we get $\tau^u(e_2) = 9$. Similarly, we get $\tau^u(e_1) = 3$ and $\tau^u(e_3) = 3$. Although these bounds are looser than the respective tight ones, they still provide a good order of visiting the entries and they can be used for pruning and checking for termination. In Section 8, we demonstrate the significant computation savings by this lightweight counting (of $\tau^u(e)$) over exact counting (of $\tau(e^-)$) and show that it affects very little the pruning power of the algorithm. Next, we investigate its effectiveness by a theoretical analysis.

[2] Note that F is the skyline of a specific data subset.

4.3 Analytical Study

Consider a dataset $\mathcal{D}$ with N points, indexed by an aR-tree whose nodes have an average fanout f. Our analysis is based on the assumption that the data points are uniformly and independently distributed in the domain space $[0,1]^d$, where d is the dimensionality. Then, the tree height h and the number of nodes n_i at level i (let the leaf level be 0) can be estimated by $h = 1 + \lceil \log_f(N/f) \rceil$ and $n_i = N/f^{i+1}$. Besides, the extent (i.e., length of any 1D projection) λ_i of a node at the i-th level can be approximated by $\lambda_i = (1/n_i)^{1/d}$ [30].

We now discuss the trade-off of lightweight counting over exact counting for a non-leaf entry e. Recall that the *exact* upper bound score $\tau(e^-)$ is counted as the number of points dominated by its lower corner e^-. On the other hand, lightweight counting obtains $\tau^u(e)$; an upper bound of $\tau(e^-)$. For a given e^-, Figure 6 shows that the space can be divided into three regions, with respect to nodes at level i. The gray region M_2 corresponds to the maximal region, covering nodes (at level i) that are *partially* dominated by e^-. While computing $\tau(e^-)$, only the entries which are *completely inside* M_2 need to be further examined (e.g., e_A). Other entries are pruned after either disregarding their aggregate values (e.g., e_B, which intersects M_1), or adding these values to $\tau(e^-)$ (e.g., e_C, which intersects M_3).

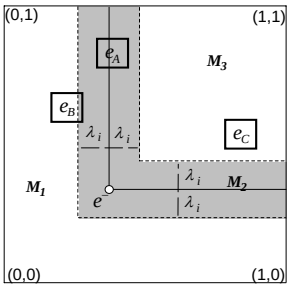


Fig. 6 I/O cost of computing upper bound

Thus, the probability of accessing a (i-th level) node can be approximated by the area of M_2, assuming that tree nodes at the same level have no overlapping. To further simplify our analysis, suppose that all coordinates of e^- are of the same value v. Hence, the aR-tree node accesses required for computing the exact $\tau(e^-)$ can be expressed as[3]:

$$NA_{exact}(e^-) = \sum_{i=0}^{h-1} n_i \cdot [(1 - v + \lambda_i)^d - (1 - v - \lambda_i)^d] \quad (3)$$

[3] For simplicity, the equation does not consider the boundary effect (i.e., v is near the domain boundary). To capture the boundary effect, we need to bound the terms $(1 - v + \lambda_i)$ and $(1 - v - \lambda_i)$ within the range $[0, 1]$.

In the above equation, the quantity in the square brackets corresponds to the volume of M_2 (at level i) over the volume of the universe (this equals to 1), capturing thus the probability of a node at level i to be completely inside M_2. The node accesses of lightweight computation can also be captured by the above equation, except that no leaf nodes (i.e., at level 0) are accessed. As there are many more leaf nodes than non-leaf nodes, lightweight computation incurs significantly lower cost than exact computation.

Now, we compare the scores obtained by exact computation and lightweight computation. The exact score $\tau(e^-)$ is determined by the area dominated by e^-:

$$\tau(e^-) = N \cdot (1 - v)^d \quad (4)$$

In addition to the above points, lightweight computation counts also all points in M_2 for the leaf level into the upper bound score:

$$\tau^u(e) = N \cdot (1 - v + \lambda_0)^d \quad (5)$$

Summarizing, three factors N, v, and d affect the relative tightness of the lightweight score bound over the exact bound.

- When N is large, the leaf node extent λ_0 is small and thus the lightweight score is tight.
- If v is small, i.e., e^- is close to the origin and has high dominating power, then λ_0 becomes less significant in Equation 5 and the ratio of $\tau^u(e)$ to $\tau(e^-)$ is close to 1 (i.e., lightweight score becomes relatively tight).
- As d increases (decreases), λ_0 also increases (decreases) and the lightweight score gets looser (tighter).

In practice, during counting-guided search, entries close to the origin have higher probability to be accessed than other entries, since their parent entries have higher upper bounds and they are prioritized by search. As a result, we expect that the second case above will hold for most of the upper bound computations and lightweight computation will be effective.

5 Priority-Based Traversal

In this section, we present a *lazy* alternative to the counting-guided method. Instead of computing upper bounds of visited entries by explicit counting, we defer score computations for entries, but maintain lower and upper bounds for them as the tree is traversed. Score bounds for visited entries are gradually refined when more nodes are accessed, until the result is finalized with the help of them. For this method to be effective, the tree is traversed with a carefully-designed priority order aiming at minimizing I/O cost. We present the basic algorithm, analyze the issue of setting an appropriate order for visiting nodes, and discuss its implementation.

5.1 The Basic Algorithm

Recall that counting-guided search, presented in the previous section, may access some aR-tree nodes more than once due to the application of counting operations for the visited entries. For instance in Figure 5, the node pointed by e_1 may be accessed twice; once for counting the scores of points under e_2 and once for counting the scores of points under e_1. We now propose a top-k dominating algorithm which traverses each node at most once and has reduced I/O cost.

Algorithm 4 shows the pseudo-code of this Priority-Based Tree Traversal Algorithm (PBT). PBT browses the tree, while maintaining (loose) upper $\tau^u(e)$ and lower $\tau^l(e)$ score bounds for the entries e that have been seen so far. The nodes of the tree are visited based on a *priority order*. The issue of defining an appropriate ordering of node visits will be elaborated later. During traversal, PBT maintains a set S of visited aR-tree entries. An entry in S can either: (i) lead to a potential result, or (ii) be partially dominated by other entries in S that may end up in the result. W is a min-heap, employed for tracking the top-k points (in terms of their τ^l scores) found so far, whereas γ is the lowest score in W (used for pruning).

First, the root node is loaded, and its entries are inserted into S after upper score bounds have been derived from information in the root node. Then (Lines 8-18), while S contains non-leaf entries, the non-leaf entry e_z with the highest priority is removed from S, the corresponding tree node Z is visited and (i) the τ^u (τ^l) scores of existing entries in S (partially dominating e_z) are refined using the contents of Z, (ii) τ^u (τ^l) values for the contents of Z are computed and, in turn, inserted to S. Note that for operations (i) and (ii), only information from the current node and S is used; no additional accesses to the tree are required. Updates and computations of τ^u scores are performed incrementally with the information of e_z and entries in S that partially dominate e_z. W is updated with points/entries of higher τ^l than γ. Finally (Line 20), entries are pruned from S if (i) they cannot lead to points that may be included in W, and (ii) are not partially dominated by entries leading to points that can reach W.

It is important to note that, at Line 21 of PBT, all non-leaf entries have been removed from the set S, and thus (result) points in W have their exact scores found.

To comprehend the functionality of PBT consider again the top-1 dominating query on the example of Figure 5. For the ease of discussion, we denote the score bounds of an entry e by the interval $\tau_\star(e)$=$[\tau^l(e), \tau^u(e)]$. Initially, PBT accesses the root node and its entries are inserted into S after their lower/upper bound scores are derived (see Lines 5–6); $\tau_\star(e_1)$=$[0, 3]$, $\tau_\star(e_2)$=$[0, 9]$, $\tau_\star(e_3)$=$[0, 3]$. Assume for now, that visited nodes are prioritized (Lines 9-10) based on the upper bound scores $\tau^u(e)$ of entries $e \in S$. Entry e_2, of the highest score τ^u in S is removed and its child node Z is accessed. Since $e_1^- \not\succ e_2^+$ and $e_3^- \not\succ e_2^+$, the upper/lower

Algorithm 4 Priority-Based Tree Traversal Algorithm (PBT)

algorithm PBT(Tree R, Integer k)
1: S:=new set; ▷ entry format in S: $\langle e, \tau^l(e), \tau^u(e)\rangle$
2: W:=new min-heap; ▷ k points with the highest τ^l
3: γ:=0; ▷ the k-th highest τ^l score found so far
4: **for all** $e_x \in R.root$ **do**
5: $\tau^l(e_x)$:=$\sum_{e\in R.root \wedge e_x^+ \succ e^-}$ COUNT(e);
6: $\tau^u(e_x)$:=$\sum_{e\in R.root \wedge e_x^- \succ e^+}$ COUNT(e);
7: insert e_x into S and update W;
8: **while** S contains non-leaf entries **do**
9: remove e_z: non-leaf entry of S with the highest *priority*;
10: read the child node Z pointed by e_z;
11: **for all** $e_y \in S$ such that $e_y^+ \not\succ e_z^- \wedge e_y^- \succ e_z^+$ **do**
12: $\tau^l(e_y)$:=$\tau^l(e_y) + \sum_{e\in Z \wedge e_y^+ \succ e^-}$ COUNT(e);
13: $\tau^u(e_y)$:=$\tau^l(e_y) + \sum_{e\in Z \wedge e_y^+ \not\succ e^- \wedge e_y^- \succ e^+}$ COUNT(e);
14: S_z:=$Z \cup \{e \in S \mid e_z^+ \not\succ e^- \wedge e_z^- \succ e^+\}$;
15: **for all** $e_x \in Z$ **do**
16: $\tau^l(e_x)$:=$\tau^l(e_z) + \sum_{e\in S_z \wedge e_x^+ \succ e^-}$ COUNT(e);
17: $\tau^u(e_x)$:=$\tau^l(e_x) + \sum_{e\in S_z \wedge e_x^+ \not\succ e^- \wedge e_x^- \succ e^+}$ COUNT(e);
18: insert all entries of Z into S;
19: update W (and γ) by $e' \in S$ whose score bounds changed;
20: remove entries e_m from S where $\tau^u(e_m) < \gamma$ and $\neg\exists e \in S,\ (\tau^u(e) \geq \gamma) \wedge (e^+ \not\succ e_m^- \wedge e^- \succ e_m^+)$;
21: **report** W as the result;

score bounds of remaining entries $\{e_1, e_3\}$ in S will not be updated (the condition of Line 11 is not satisfied). The score bounds for the points p_1, p_2, and p_3 in Z are then computed; $\tau_\star(p_1)$=[1, 7], $\tau_\star(p_2)$=[0, 3], and $\tau_\star(p_3)$=[0, 3]. These points are inserted into S, and W=$\{p_1\}$ with γ=$\tau^l(p_1)$=1. No entry or point in S can be pruned, since their upper bounds are all greater than γ. The next non-leaf entry to be removed from S is e_1 (the tie with e_3 is broken arbitrarily). The score bounds of the existing entries S=$\{e_3, p_1, p_2, p_3\}$ are in turn refined; $\tau_\star(e_3)$ remains $[0, 3]$ (unaffected by e_1), whereas $\tau_\star(p_1)$=[3, 6], $\tau_\star(p_2)$=[1, 1], and $\tau_\star(p_3)$ =[0, 3]. The scores of the points indexed by e_1 are computed; $\tau_\star(p_4)$=[0, 0], $\tau_\star(p_5)$=[0, 0], and $\tau_\star(p_6)$=[1, 1] and W is updated to p_1 with γ=$\tau^l(p_1)$=3. At this stage, all points, except from p_1, are pruned from S, since their τ^u scores are at most γ and they are not partially dominated by non-leaf entries that may contain potential results. Although no point from e_3 can have higher score than p_1, we still have to keep e_3, in order to compute the exact score of p_1 in the next round.

5.2 Traversal Orders in PBT

An intuitive method for prioritizing entries at Line 9 of PBT, hinted by the *upper bound principle* of [19] or the *best-first ordering* of [13, 24], is to pick the entry e_z with the highest upper bound score $\tau^u(e_z)$; such an order would visit the points that have high probability to be in the top-k dominating result early. We denote this instantiation of PBT by UBT (for Upper-bound Based Traversal).

Nevertheless a closer look into PBT (Algorithm 4) reveals that the upper score bounds alone may not offer the best priority order for traversing the tree. Recall that the pruning operation (at Line 20) eliminates entries from S, saving significant I/O cost and leading to the early termination of the algorithm. The effectiveness of this pruning depends on the *lower* bounds of the best points (stored in W). Unless these bounds are tight enough, PBT will not terminate early and S will grow very large.

For example, consider the application of UBT to the tree of Figure 2. The first few nodes accessed are in the order: root node, e_{18}, e_{11}, e_9, e_{12}. Although e_{11} has the highest upper bound score, it *partially dominates* high-level entries (e.g., e_{17} and e_{20}), whose child nodes have not been accessed yet. As a result, the best-k score γ (i.e., the current lower bound score of e_{11}) is small, few entries can be pruned, and the algorithm does not terminate early.

Thus, the objective of search is not only to (i) examine the entries of large upper bounds early, which leads to early identification of candidate query results, but also (ii) eliminate partial dominance relationships between entries that appear in S, which facilitates the computation of tight lower bounds for these candidates. We now investigate the factors affecting the probability that one node partially dominates another and link them to the traversal order of PBT. Let a and b be two random nodes of the tree such that a is at level i and b is at level j. Using the same uniformity assumptions and notation as in Section 4.3, we can infer that the two nodes a and b not intersect along dimension t with probability[4]:

$$Pr(a[t] \cap b[t] = \varnothing) = 1 - (\lambda_i + \lambda_j)$$

a and b have a partial dominance relationship when they intersect along at least one dimension. The probability of being such is:

$$Pr(\bigvee_{t\in[1,d]} a[t] \cap b[t] \neq \varnothing) = 1 - (1 - (\lambda_i + \lambda_j))^d$$

The above probability is small when the sum $\lambda_i + \lambda_j$ is minimized (e.g., a and b are both at low levels).

The above analysis leads to the conclusion that in order to minimize the partially dominating entry pairs in S, we should prioritize the visited nodes based on their level at the tree. In addition, between entries at the highest level in S, we should choose the one with the highest upper bound, in order to find the points with high scores early. Accordingly, we propose an instantiation of PBT, called Cost-Based Traversal (CBT). CBT corresponds to Algorithm 4, such that,

[4] The current equation is simplified for readability. The probability equals 0 when $\lambda_i + \lambda_j > 1$.

at Line 9, the non-leaf entry e_z with the highest level is removed from S and processed; if there are ties, the entry with the highest upper bound score is picked. In Section 8, we demonstrate the advantage of CBT over UBT in practice.

5.3 Implementation Details

A straightforward implementation of PBT may lead to very high computational cost. At each loop, the burden of the algorithm is the pruning step (Line 20 of Algorithm 4), which has worst-case cost quadratic to the size of S; entries are pruned from S if (i) their upper bound scores are below γ and (ii) they are not partially dominated by any other entry with upper bound score above γ. If an entry e_m satisfies (i), then a scan of S is required to check (ii).

In order to check for condition (ii) efficiently, we use a main-memory R-tree $I(S)$ to index the entries in S having upper bound score above γ. When the upper bound score of an entry drops below γ, it is removed from $I(S)$. When checking for pruning of e_m at Line 20 of PBT, we only need to examine the entries indexed by $I(S)$, as only these have upper bound scores above γ. In particular, we may not even have to traverse the whole index $I(S)$. For instance, if a non-leaf entry e' in $I(S)$ does not partially dominate e_m, then we need not check for the subtree of e'. As we verified experimentally, maintaining $I(S)$ enables the pruning step to be implemented efficiently. In addition to $I(S)$, we tried additional data structures for accelerating the operations of PBT (e.g., a priority queue for popping the next entry from S at Line 9), however, the maintenance cost of these data structures (as the upper bounds of entries in S change frequently at Lines 11-13) did not justify the performance gains by them.

6 Query Processing on Non-indexed Data

This section examines the evaluation of top-k dominating queries on non-indexed data, assuming that data points are stored in random order in a disk file $\mathcal{D}$.

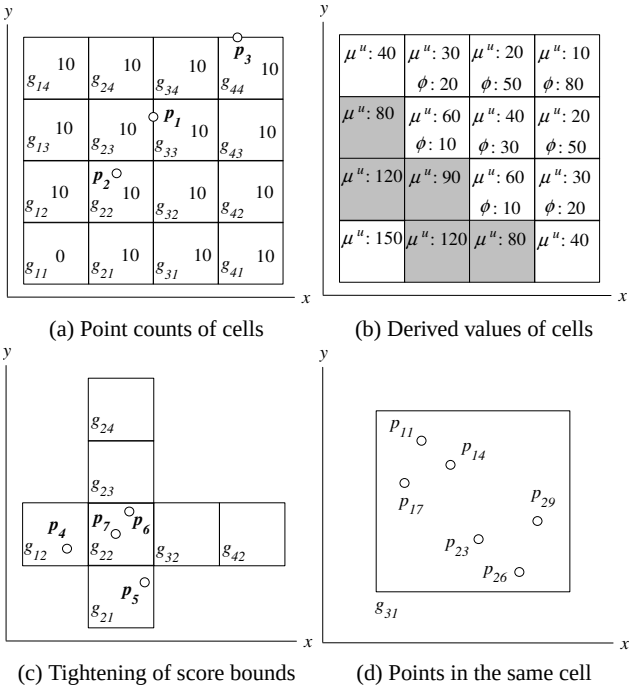
As discussed in [31], a practically viable solution is to first bulk-load an aR-tree (e.g., using the algorithm of [18]) from the dataset and then compute top-k dominating points using the algorithms proposed in Sections 4 and 5. The bulk-loading step requires externally sorting the points, which is known to scale well for large datasets. However, external sorting may incur multiple I/O passes over data.

Our goal is to compute the top-k dominating points with only a constant number (3) of data passes, by adopting the filter-refinement framework. The first pass is the *counting pass*, which employs a memory grid structure to keep track of point count in cells, while scanning over the data. This structure is then used to derive lower/upper bound scores of points in the next pass. The second pass is the *filter pass*, which applies pruning rules to discard unqualified points and keep the remaining ones in a candidate set. The *refinement pass*, being the final pass, performs a scan over the data in order to count the exact τ scores of all candidate points. Eventually, the top-k dominating points are returned.

In Section 6.1, we present the details of the counting pass. We investigate different techniques for the filter pass in Sections 6.2 and 6.3; these techniques trade-off efficiency (i.e., CPU time at the filter step) for filter effectiveness (i.e., size of the candidate set). Finally, Section 6.4 discusses the final, refinement pass of the algorithm.

6.1 The counting pass

The first step of the algorithm defines a regular multi-dimensional grid over the space and performs a linear scan to the data to count the number of points in each grid cell. Such a 2-dimensional histogram (with 4×4 cells) is shown in Figure 7a. To ease our discussion, each grid cell is labeled as g_{ij}. While scanning the points, we increase the counters of the cells that contain them, but do not keep the visited points in memory. In this example, at the end of scan, we have $\texttt{COUNT}(g_{11})$=0 and $\texttt{COUNT}(g_{12})$=10. We adopt the following convention so that each point contributes to the counter of exactly one cell. In case a point (e.g., p_1) falls on the common border of multiple cells (e.g., g_{23} and g_{33}), it belongs to the cell (e.g., g_{33}) with the largest coordinates.



(a) Point counts of cells (b) Derived values of cells

(c) Tightening of score bounds (d) Points in the same cell

Fig. 7 Using the grid in the filter step, k=2

After the counting pass, and before the filter pass begins, we can derive lower/upper bound scores of the cells from their point counts, by using the notations of Section 3.1. This enables us to determine fast the cells that cannot contain top-k dominating points. Given a grid cell g, its upper bound score $\tau^u(g)$ is the total point count of cells it partially or totally dominates.

$$\tau^u(g) = \sum_{g_y \in \mathcal{G} \wedge g^- \succ g_y^+} \mathsf{COUNT}(g_y)$$

In Figure 7b, the cell g_{33} dominates g_{33}, g_{43}, g_{34}, and g_{44}, so we have $\tau^u(g_{33})$=40. The lower bound score $\tau^l(g)$ of g is the total point count of cells it *fully* dominates.

$$\tau^l(g) = \sum_{g_y \in \mathcal{G} \wedge g^+ \succ g_y^-} \mathsf{COUNT}(g_y)$$

For instance, g_{33} fully dominates g_{44} so we obtain $\tau^l(g_{33})$=10 (not shown in the figure).

Besides score bounds, pruning can also be achieved with the help of the dominance property. From Equation 2, we observe that, a point cannot belong to the result if it is dominated by k other points. Thus, we define the dominated count $g.\phi$ of the cell g as the total point count of cells fully dominating g.

$$g.\phi = \sum_{g_y \in \mathcal{G} \wedge g_y^+ \succ g^-} \mathsf{COUNT}(g_y)$$

For example, g_{32} is fully dominated by g_{11} and g_{21}, so we get $g_{32}.\phi$=0+10=10. Clearly, a cell with $g_{ij}.\phi \geq k$ cannot contain any of the top-k results.

Let $k = 2$ in the example of Figure 7b. We proceed to determine the cells that cannot contain query results. These include cells with zero count (e.g., g_{11}) and cells having $g_{ij}.\phi \geq k$ (e.g., g_{23}). In order to obtain the value γ (lower bound score of top-k points), we enumerate the remaining cell(s) in descending order of their τ^l scores, until their total point count reaches k. Since the cell g_{12} contains 10 ($\geq k$) points and its τ^l score is 60, we set $\gamma = 60$. Obviously, cells (e.g., g_{14}) whose upper score bounds below $\gamma = 60$ can be pruned. The remaining cells (containing potential results) are colored as gray in Figure 7b.

6.2 Coarse-grained Filter

During, the second (filter) pass, the algorithm scans the data again and determines a set of candidate points for the top-k dominating query. The first method we propose for the filter pass is called *coarse-grained filter* (CRS). CRS scans the database and uses the score bounds of grid cells and the dominance property (of Equation 2) to prune points. CRS is described by Algorithm 5. Each cell g is coupled with a candidate set $g.C$, for maintaining candidate points that fall in g (this is done only for cells g that are not pruned after the counting pass). Initially, we have no information about the detailed contents of the cells. However, using the lower score bounds τ^l of the cells and their cardinalities, we can initialize γ; the k-th highest τ^l score of the top-k dominating candidates. In other words, we assume that each candidate has the maximum coordinates in its container cell g (worst case) and use $\tau^l(g)$ as its lower bound. The algorithm then performs a linear scan over the dataset $\mathcal{D}$ at Lines 4–16. For the point p being currently examined, we initialize its upper bound $\tau^u(p)$ and *dominated count* $p.\phi$ using the corresponding values of its container cell g_p.

Algorithm 5 Coarse-grained Filter Algorithm (CRS-Filter)

```
    algorithm CRS-Filter(Dataset D, Integer k, Grid G)
 1: for all cell g ∈ G do
 2:     g.C:=new set;                          ▷ candidate set of the cell
 3: γ:=the k-th highest τ^l score of cells in G;   ▷ for each cell g ∈ G,
    COUNT(g) instances of the score τ^l(g) are considered
 4: for all p ∈ D do                                          ▷ filter scan
 5:     let g_p be the grid cell of p;
 6:     τ^u(p):=τ^u(g_p); p.φ:=g_p.φ;
 7:     for all cells g_z ∈ G such that g_z^- ≻ g_p^+ ∧ g_z^+ ⊁ g_p^- do
 8:         for all p' ∈ g_z.C such that p' ≻ p do
 9:             p.φ:=p.φ + 1;
10:             if p.φ ≥ k then
11:                 ignore further processing for the point p;
12:     for all cells g_z ∈ G such that g_p^- ≻ g_z^+ ∧ g_p^+ ⊁ g_z^- do
13:         for all p' ∈ g_z.C such that p ≻ p' do
14:             p'.φ:=p'.φ + 1;
15:             if p'.φ ≥ k then
16:                 remove p' from g_z.C;
17:     if τ^u(p) ≥ γ and p.φ < k then
18:         insert p into g_p.C;
```

In the loop of Lines 7–11, we search for candidate points p' that dominate p and have already been read in memory. For each such occurrence, the value $p.\phi$ is incremented. Due to the presence of the dominated count $g_p.\phi$ of the grid cell, it suffices to traverse only the cells that partially dominate the cell of p (as opposed to all cells). Whenever $p.\phi$ reaches k, $p.\phi$ needs not be incremented further (and the loop exits); in this case p cannot be a top-k dominating result.

In the loop of Lines 12–16, we search for candidate points p' that are dominated by p and have already been read in memory. The $p'.\phi$ of each such point p' is incremented; and the point is pruned from the candidate set when $p'.\phi$ reaches k. Note that Lines 12–16 need not be executed when $p.\phi$ is (at least) k. The reason is that any existing candidate p' which is dominated by p must have also been dominated by the k dominators of p and therefore already been pruned in a previous iteration.

At Lines 15–16, we insert the current point p into the candidate set $g_p.C$ of its cell g_p, only when its $\tau^u(p)$ score is above γ and its $p.\phi$ value is less than k.

6.3 Fine-grained Filter

CRS simply sets the score bounds of candidate points to those of their cells. Since each cell may contain a large number of points, their score bounds are not tight, weakening the filter effectiveness of CRS. In this section, we develop a fine-grained solution (FN) that tightens the score bounds of candidate points gradually. This way, more unqualified points having low scores can be eliminated from the search early.

Tightening the score bounds of points. Consider the filter step during the processing of the top-2 dominating query (i.e., k=2) on Figure 7c. Suppose that, the points p_4, p_5, p_6 are existing candidates (they have already been read during the filter pass), and the next point to be processed is p_7.

The first technique is to tighten score bounds by using the current point p_7 and existing candidate points p_4, p_5, p_6. First of all, we set $\tau^l(p_7) = 40$ and $\tau^u(p_7) = 90$, by using score bounds of p_7's cell g_{22}. To tighten score bounds of existing candidates, we traverse the cells (i.e., g_{12}, g_{22}, g_{21}) that partially dominate g_{22}. Since p_4 dominates p_7, we increment $\tau^l(p_4)$. On the other hand, p_5 and p_6 do not dominate p_7 so their τ^u scores are decremented. To tighten score bounds of the current point p_7, we traverse the cells (e.g., g_{22}, g_{32}, g_{42}, g_{23}, g_{24}) that are partially dominated by g_{22}. As p_7 dominates p_6, we increment $\tau^l(p_7)$. In addition, the dominated count of p_6 now becomes 2 ($\geq k$) so it is removed from the local candidate set $g_{22}.C$.

A second technique that our filter algorithm uses to tighten score bounds is by utilizing bounds of candidate points that have not already been pruned. Assume in Figure 7d that, the point p_{17} is visited after points p_{11} and p_{14} (intermediate points like p_{12} have been pruned). In this case, $\tau^l(p_{17})$ can be tightened to $\max\{\tau^l(p_{17}), \tau^l(p_{11}), \tau^l(p_{14})\}$. As another example, suppose that the point p_{29} is visited after points p_{23} and p_{26}. Then, the upper bound score of p_{29} can be tightened to $\min\{\tau^u(p_{29}), \tau^u(p_{23}), \tau^u(p_{26})\}$.

Writing disk partitions. We observe that the pruning effectiveness of the algorithm can be significantly improved if we are able to identify points with high scores early. To achieve this, we modify the counting pass (described in Section 6.1) as follows. Each grid cell g is allocated a memory partition (at least one page) to store the accessed points that fall in the cell. Whenever the memory becomes full, the largest memory partition is flushed into its corresponding disk partition $g.\mathcal{D}$ (i.e., a sequential file). At the end of the counting pass, remaining points in memory are flushed into their respective disk partitions. This modification costs an additional writing pass over the data, yet it permits us to access the disk partitions using different orderings (in the subsequent filter and refinement passes).

Algorithm. Algorithm 6 presents the details of our Fine-grained Filter Algorithm (FN-Filter). A min-heap W is used to keep track of k points with the highest τ^l scores seen so far and γ is set to the k-th score in W. Like in the CRS-Filter, we first determine the k-th highest lower bound score γ from the τ^l scores and point counts of grid cells. Then, k dummy pairs having the score γ are inserted into W. The set S contains the grid cells whose disk partitions have yet to be visited. Initially, all grid cells are inserted into S.

Algorithm 6 Fine-grained Filter Algorithm (FN-Filter)

algorithm FN-Filter(Dataset $\mathcal{D}$, Integer k, Grid $\mathcal{G}$)
1: γ:=the k-th highest τ^l score of cells in $\mathcal{G}$; ▷ for each cell $g \in \mathcal{G}$, COUNT(g) instances of the score $\tau^l(g)$ are considered
2: W:=new min-heap; ▷ k points with the highest τ^l
3: insert k dummy pairs $\langle NULL, \gamma\rangle$ into W;
4: S:=new set; ▷ set of grid cells
5: **for all** cell $g \in \mathcal{G}$ **do**
6: $g.C$:=new set; ▷ candidate set of the cell
7: let $g.\mathcal{D}$ be the disk partition of g;
8: insert g into S;
9: **while** S is non-empty **do**
10: remove g: the cell in S with the highest *priority*;
11: **if** $\tau^u(g) < \gamma$ and $\neg\exists g_z \in \mathcal{G},\ (\tau^u(g_z) \geq \gamma) \wedge (g_z^+ \not\succ g^- \wedge g_z^- \succ g^+)$ **then**
12: ignore further processing for the disk partition $g.\mathcal{D}$;
13: **for all** $p \in g.\mathcal{D}$ **do** ▷ scan over points in disk partition $g.\mathcal{D}$
14: $\tau^l(p)$:=$\tau^l(g)$; $\tau^u(p)$:=$\tau^u(g)$; $p.\phi$:=$g.\phi$;
15: $\delta.l$:=-1; $\delta.u$:=$|\mathcal{D}|$; $\delta.\phi$:=-1;
16: **for all** cell $g_z \in \mathcal{G}$ such that $g_z^- \succ g^+ \wedge g_z^+ \not\succ g^-$ **do**
17: **for all** $p' \in g_z.C$ **do** ▷ existing candidates in memory
18: **if** $p' \succ p$ **then**
19: $\tau^l(p')$:=$\tau^l(p') + 1$; $p.\phi$:=$p.\phi + 1$;
20: $\delta.u$:=$\min\{\ \delta.u,\ \tau^u(p')\ \}$;
21: $\delta.\phi$:=$\max\{\ \delta.\phi,\ p'.\phi\ \}$;
22: **else**
23: $\tau^u(p')$:=$\tau^u(p') - 1$;
24: **for all** cell $g_z \in \mathcal{G}$ such that $g^- \succ g_z^+ \wedge g^+ \not\succ g_z^-$ **do**
25: **for all** $p' \in g_z.C$ **do** ▷ existing candidates in memory
26: **if** $p \succ p'$ **then**
27: $\tau^l(p)$:=$\tau^l(p) + 1$; $p'.\phi$:=$p'.\phi + 1$;
28: $\delta.l$:=$\max\{\ \delta.l,\ \tau^l(p')\ \}$;
29: **else**
30: $\tau^u(p)$:=$\tau^u(p) - 1$;
31: $\tau^l(p)$:=$\max\{\ \tau^l(p),\ \delta.l\ \}$; $\tau^u(p)$:=$\min\{\ \tau^u(p),\ \delta.u\ \}$;
32: $p.\phi$:=$\max\{\ p.\phi,\ \delta.\phi + 1\ \}$;
33: **if** $\tau^l(p) > \gamma$ and $p.\phi < k$ **then**
34: update W (and γ), by $\langle p, \tau^l(p)\rangle$;
35: **if** $\tau^u(p) \geq \gamma$ and $p.\phi < k$ **then**
36: insert p into $g.C$;
37: update W (and γ) by points whose τ^l scores $> \gamma$;
38: remove points $p'' \in g_y.C$ (where $g_y \in \mathcal{G}$) satisfying the condition $p''.\phi \geq k$ or $\tau^u(p'') < \gamma$;

At Line 10, we pick the grid cell g from S with the highest *priority* value, which will be elaborated shortly. In case the cell has upper bound score $\tau^u(g)$ below γ and it is not partially dominated by any other grid cell g_z with $\tau^u(g_z) \geq \gamma$, the disk partition $g.\mathcal{D}$ of g is ignored. The reason is that (i) g may not contain any top-k point and (ii) its contribution to top-k candidates has already been captured in their upper/lower bounds. Otherwise, at Lines 9–38, a scan is performed over the points in $g.\mathcal{D}$. At Line 14,

we set the score bounds and dominated count of the current point p to that of its cell g_p. At Lines 16–23, we traverse the candidates in the cells that are partially dominating g_p in order to update score bounds. This is done only for cells whose partitions have been loaded before. Similarly, at Lines 24–30, we traverse the candidates in cells partially dominated by g_p, in order to tighten the score bounds of the current point p. Meanwhile, we record the value of: (i) $\delta.l$, the maximum τ^l score of points dominated by p (ii) $\delta.u$, the minimum τ^u score of points dominating p, and (iii) $\delta.\phi$, the maximum dominated count $p'.\phi$ of points p' dominating p. These values are then used to update the score bounds and the dominated count of the current point p. In case $\tau^l(p)$ is greater than γ, we update the top-k points in W. If $\tau^u(p)$ is at least γ, then we insert p into the local candidate set of its grid cell. At Lines 37–38, existing candidate points having τ^l scores above γ are used to update W, and points with τ^u scores below γ are pruned.

Order of searching disk partitions. We now investigate concrete orderings for accessing disk partitions, at Line 10 of the FN-Filter algorithm. We first suggest the *scanline ordering* as a reference, which accesses cells g in ascending order of the value: $SLV(g) = \sum_{i=1}^{d}(T_i(g)-1)\cdot A^{i-1}$ where A is the number of divisions per dimension and $T_i(g) = A \cdot g[i]^+/\varsigma$ (assuming domain as $[0,\varsigma]^d$). For instance, the value of A is 4 in Figure 7a, and we have $SLV(g_{31}) = (3-1)\cdot 1 + (1-1)\cdot 4 = 2$. Disk partitions of cells are visited in the order: g_{11}, g_{21}, g_{31}, g_{41}, g_{12}, g_{22}, $\cdots$. This ordering is independent of score bounds of cells.

Another ordering we consider is the *upper bound score ordering*, which visits the cells in descending order of their upper bound scores. In the example of Figure 7b, the cells will be visited in the order: g_{11}, g_{21}, g_{12}, g_{22}, $\cdots$. This ordering allows us to identify early points with high scores. However, it may delay accessing cells that have low upper bound, but partially dominate those with high upper bounds. This delays the tightening of loose bounds and, in turn, the pruning of points.

Finally, we investigate a *partial dominance elimination ordering*, which takes partial dominance relationships among the partitions into account. We pick the cell (say, g_a) with the highest upper bound score, that partially dominates some unvisited cells. In case g_a has not been visited before, we access its disk partition. Then, we access partitions of all unvisited cells g_b that are partially dominated by g_a, in descending order of their upper bound scores. The above procedure repeats until the cells are exhausted. For instance, in Figure 7b, we first visit the cell g_{21}, and then visit the cells partially dominated by it in descending upper bound score order: g_{22}, g_{31}, g_{23}, g_{41}, g_{24}. Next, we visit the cell g_{12}, and unseen cells partially dominated by it: g_{13}, g_{32}, g_{14}, g_{42}.

According to these orderings, we denote the instantiations of the fine-grained method as follows: FNS (with scanline ordering), FNU (with upper bound score ordering), and FNP (with partial dominance elimination ordering).

6.4 The refinement pass

After completing the filter pass, we obtain a set C of candidate points, which have potential to be the actual results. In the refinement pass, a linear scan is performed over the dataset $\mathcal{D}$; each point $p' \in \mathcal{D}$ is compared against each candidate $p \in C$ and the score of p is incremented when p dominates p'. This straightforward implementation requires $|\mathcal{D}| \cdot |C|$ dominance comparisons and becomes expensive even for moderate-sized candidate set.

In order to accelerate the refinement pass, we take advantage of the lower score bounds of grid cells. Suppose that p_7 is a candidate point in Figure 7c. Since it falls in the cell g_{22}, we set the lower bound score of p_7 to $\tau^l(p_7) = \tau^l(g_{22}) = 40$. While scanning over $\mathcal{D}$ in the refinement pass, we need not compare each point $p' \in \mathcal{D}$ with the candidate p_7. Only points p' in cells that are partially dominated by g_{22} (i.e., g_{22}, g_{32}, g_{42}, g_{23}, g_{24}) have to be compared with p_7.

Algorithm 7 is the pseudo-code of the grid-based refinement algorithm. $\mathcal{G}$ represents the grid obtained from the counting pass. Each grid cell $g \in \mathcal{G}$ is associated with a local candidate set $g.C$, for storing candidates (from the filter pass) that falls into the cell g. The value γ is set to the k-th highest τ^l score of all candidates (assuming that their score bounds are obtained from the filter pass). At Line 3, we check if a cell g has τ^u score below γ and it is not partially dominated by any cell g_z having some candidate point. If so, the cell is marked as *irrelevant* as it cannot influence the top-k result. At Line 5, the lower bound score $\tau^l(p)$ of each candidate $p \in g.C$ is reset to $\tau^l(g)$. Then, a scan is performed over the dataset $\mathcal{D}$. In case the cell $g_{p'}$ of the current point $p' \in \mathcal{D}$ is *irrelevant*, the point p' is discarded immediately without further processing. At Lines 11–13, only the cells partially dominating p' need to be considered. Every candidate p in such a cell is compared with p', and its score $\tau^l(p)$ is incremented when p dominates p'. Eventually, the k candidate points with highest scores are returned as the query results.

The above refinement algorithm is generic in the sense that it does not utilize disk partitions of cells (created in FN-Filter). To optimize its performance, we replace the linear scan at Line 7 by a promising order of accessing disk partitions of cells (e.g., starting with partitions that are partially dominated by the candidate with the highest upper bound score). Nevertheless, this optimization technique cannot be applied if the filter step is performed by the CRS-Filter, which does not build disk partitions of points for cells.

Algorithm 7 Grid-based Refinement Algorithm

algorithm GridRefinement(Dataset $\mathcal{D}$, Integer k, Grid $\mathcal{G}$)
1: γ:=the k-th highest τ^l score of candidates in cells of $\mathcal{G}$;
2: **for all** cell $g \in \mathcal{G}$ **do**
3: **if** $\tau^u(g) < \gamma$ and $\neg\exists g_z \in \mathcal{G},\ (|g_z.C| > 0) \wedge (g_z^+ \not\succ g^- \wedge g_z^- \succ g^+)$ **then**
4: mark the cell g as *irrelevant*;
5: **for all** $p \in g.C$ **do**
6: $\tau^l(p)$:=$\tau^l(g)$; ▷ reset lower bound score
7: **for all** $p' \in \mathcal{D}$ **do**
8: let $g_{p'}$ be the grid cell of p';
9: **if** $g_{p'}$ is *irrelevant* **then**
10: ignore further processing for point p';
11: **for all** cell $g \in \mathcal{G}$ such that $g^- \succ g_{p'}^+ \wedge g^+ \not\succ g_{p'}^-$ **do**
12: **for all** $p \in g.C$ such that $p \succ p'$ **do**
13: $\tau^l(p)$:=$\tau^l(p) + 1$;
14: **return** k points in $\bigcup_{g\in\mathcal{G}} g.C$ with the highest τ^l scores;

7 Relaxed Top-k Dominating Query

In this section, we study a relaxed variant of the top-k dominating query. Section 7.1 presents the motivation and definition of this query. We discuss adaptations of our tree-based algorithms for evaluating this query in Sections 7.2, 7.3, 7.4.

7.1 Motivation

While the score $\tau(p)$ models nicely the intuitive importance of a point p, the dominance requirement may be too strict in particular data distributions, where all points may have similar scores. Table 1 shows the coordinate values of three points in the 3-dimensional space. Since each point does not dominate any other point in the dataset, we obtain $\tau(p_1) = \tau(p_2) = \tau(p_3) = 0$. In this case, we cannot identify the most "important" point from the dataset.

Point p	$p[1]$	$p[2]$	$p[3]$
p_1	1	2	3
p_2	3	1	4
p_3	4	3	2

Table 1 Example of points in the 3-dimensional space

To avoid this problem, we propose to relax the dominance requirement as follows. Given two points $p, p' \in \mathcal{D}$, we define the set $\omega(p, p')$ of dimensions such that p is smaller than (i.e., preferable to) p' along these dimensions:

$$\omega(p, p') = \{\ i \mid i \in [1, d]\ \wedge\ p[i] < p'[i]\ \} \quad (6)$$

Then, we define $\psi(p, p') = 2^{|\omega(p,p')|} - 1$ (i.e., the number of non-empty dimensional subsets of $\omega(p, p')$). As p dominates p' with respect to each of these $\psi(p, p')$ dimensional subsets, we define the *relaxed score* of a point p as:

$$\tau_r(p) = \sum_{p'\in\mathcal{D}} \psi(p, p')$$

The *relaxed top-k dominating query* returns k points in the dataset $\mathcal{D}$ with the highest τ_r score.

As an example, we consider τ_r scores of points in Table 1. By comparing p_1 with other points, we get $\omega(p_1, p_2)$=$\{1, 3\}$ and $\omega(p_1, p_3)$=$\{1, 2\}$. Thus, we have $\tau_r(p_1) = (2^2 - 1) + (2^2 - 1) = 6$. Similarly, we can obtain $\tau_r(p_2) = (2^1 - 1) + (2^2 - 1) = 4$ and $\tau_r(p_3) = (2^1 - 1) + (2^1 - 1) = 2$. Now, we are able to rank the three points based on their dominance scores (e.g., p_1 is the top-1 point in the dataset). In Section 8, we demonstrate that this relaxed query is appropriate for search in datasets with missing values.

Regarding the definition of $\psi(p, p')$, we use the number $2^{|\omega(p,p')|} - 1$ of dimensional subsets, as opposed to the number of dimensions in $\omega(p, p')$. The rationale is that, a point should be assigned a very high weight if it dominates others in a large number of dimensions. For example, consider two points p_1 and p_2, such that p_1 dominates 10 points, each along 10 dimensions, and p_2 dominates 9 points, each along 11 dimensions. Intuitively, although p_2 dominates fewer points, p_2 should have higher score than p_1 because more combinations of dimensions are involved in the dominance relationships. The score function $\psi(p, p')$ captures exactly this intuition. On the other hand, if the value $|\omega(p, p')|$ is used as a replacement of $\psi(p, p')$ in the definition of $\tau_r(p)$, then p_1 appears better than p_2, violating the above intuition.

It fell to our attention that the relaxed top-k dominating query shares some similarities with the concept of top-k frequent skyline points in dimensional subsets [6]. The major difference of our work from [6] is that we do not consider *skyline* points only. The dimensional subset $\omega(p, p')$ contributes to the relaxed score $\tau_r(p)$ of p, even when p is not a skyline point in $\mathcal{D}$ with respect to $\omega(p, p')$. In addition, [6] emphasizes on approximate result computation but we focus on exact evaluation of our relaxed query over aR-trees. Unlike the k-dominant skyline query [5], our relaxed query does not require any apriori value of the subspace size.

7.2 Adaptation of Skyline-Based Approach

In this section, we discuss the adaptation of the skyline-based approach (in Section 3.2) for processing the relaxed top-k dominating query. In particular, we study the modifications of the followings: (i) the dominance property of Equation 2, and (ii) the BatchCount procedure (Algorithm 1), which counts the exact scores for a set of points.

Monotone property for the relaxed score. First of all, we prove that the monotone property holds for the relaxed score τ_r as well. This property, expressed by Equation 7, is not only essential to the skyline-based approach, but also important for other tree-based solutions.

$$\forall\ p, p' \in \mathcal{D},\ p \succ p' \Rightarrow \tau_r(p) > \tau_r(p') \quad (7)$$

The proof is as follows. Consider any point $a \in \mathcal{D}$ such that $a \neq p$ and $a \neq p'$. Since p dominates p', we have $\omega(p', a) \subseteq \omega(p, a)$. As a result, a contributes to $\tau_r(p)$ at least as much as it contributes to $\tau_r(p')$. In addition to that, p' contributes zero to $\tau_r(p')$ (because p' does not dominate itself in any dimension), but p' contributes at least 1 to $\tau_r(p)$ (i.e., $\omega(p, p) \geq 1$) because $p \succ p'$. As a result, we obtain $\tau_r(p) > \tau_r(p')$.

Exact score counting. Next, we study how to compute the exact τ_r score of a point, by using the aR-tree. We proceed to present the relevant notations in the context of the relaxed score. Given two (non-leaf) entries e, e' of the tree, we define $\omega^l(e, e')$ as the minimal set of dimensions such that e always dominates e', and $\omega^u(e, e')$ as the maximal set of dimensions such that e potentially dominates e':

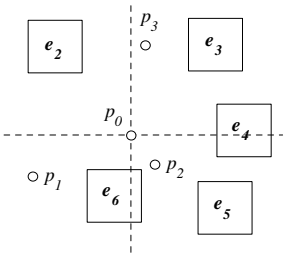
$$\omega^l(e, e') = \{\ i \mid i \in [1, d]\ \wedge\ e[i]^+ < e'[i]^-\ \}$$

$$\omega^u(e, e') = \{\ i \mid i \in [1, d]\ \wedge\ e[i]^- < e'[i]^+\ \}$$

As a shorthand notation, we define $\psi^l(e, e')$ and $\psi^u(e, e')$ as $(2^{|\omega^l(e,e')|} - 1)$ and $(2^{|\omega^u(e,e')|} - 1)$ respectively. In our subsequent discussion, these values are used to derive lower/upper bound scores for e. Note that, $\omega^l(e, e')$ and $\omega^u(e, e')$ are equal if and only if e does not intersect e' along any dimension. Otherwise, $\omega^l(e, e')$ is a proper subset of $\omega^u(e, e')$. Observe that the above notations are applicable for points p and p' as well, by replacing e by p (and e' by p').

We modify BatchCount (Algorithm 1) as follows so that it can be used to compute the τ_r values of points (instead of their τ values). First, the sub-condition $p \succ e^+ \wedge p \not\succ e^-$ at Line 2 is replaced by $\psi^u(p, e) > \psi^l(p, e)$. Second, Lines 7–8 are replaced by the statement $\tau_r(p)$:=$\tau_r(p) + \psi^l(p, e) \cdot \mathtt{COUNT}(e)$

As an example, we apply the above technique to compute the τ_r score for the point p_0 in Figure 8a, which also shows the other points/entries to be visited in the aR-tree. Initially, the value $\tau_r(p_0)$ is set to zero. The detailed steps are elaborated in Figure 8b. When a point (say, p_3) is encountered, we simply increment $\tau_r(p_0)$ by $\psi(p_0, p_3)$. The same is repeated for any non-leaf entry (say, e_2) satisfying $\psi^l(p_0, e_2)$=$\psi^u(p_0, e_2)$, except that its count value $\mathtt{COUNT}(e_2)$ is taken into account. In case a non-leaf entry (say, e_4) has different values for $\psi^l(p_0, e_4)$ and $\psi^u(p_0, e_4)$, its child node will be visited.



(a) The case of a point

Entry	Action on $\tau_r(p_0)$
p_1	add 0
p_2	add (2^1-1)
p_3	add (2^2-1)
e_2	add (2^1-1)$\mathtt{COUNT}(e_2)$
e_3	add (2^2-1)$\mathtt{COUNT}(e_3)$
e_4	visit its child node
e_5	add (2^1-1)$\mathtt{COUNT}(e_5)$
e_6	visit its child node

(b) Derivation of $\tau_r(p_0)$

Fig. 8 Exact computation of the τ_r value for a point

7.3 Adaptation of Counting-Guided Search

We proceed to elaborate the adaptation of the counting-guided search (e.g., SCG, LCG) for the relaxed query. According to Equation 7, the monotone property still holds for the relaxed score. This enables us to eliminate unqualified entries by using the pruner set (see Section 4.2). In addition, the technique for counting exact τ_r scores of points (discussed in Section 7.2) can be reused for SCG and LCG.

Recall that SCG computes upper bounds of non-leaf entries (at Line 10). Due to the monotone property of Equation 7, the tight upper bound score of an entry e is taken as $\tau_r(e^-)$. In the example of Figure 10a, the lower corner of e_1 is e_1^- and the value $\tau_r(e_1^-)$ is a tight upper bound score for any point indexed under the subtree of e_1. This value (i.e., $\tau_r(e_1^-)$) can be obtained by applying the exact counting technique described in Section 7.2.

Following the uniformity assumption and the notations from Section 4.3, we now analyze the cost of computing the exact $\tau_r(e^-)$ value for a non-leaf entry e. With respect to tree nodes at level i, the space is decomposed into two regions, as shown in Figure 9. The region M (in gray) fully contains the nodes whose parent entries e' satisfy the condition $\psi^u(e, e') > \psi^l(e, e')$; whereas the white region intersects all other nodes. By translating the area of M to the access cost, the aR-tree node accesses for computing the exact $\tau_r(e^-)$ can be expressed as:

$$NA_{exact}^{relax}(e^-) = \sum_{i=0}^{h-1} n_i \cdot [d(2\lambda_i)^{d-1} - (d-1)(2\lambda_i)^d] \quad (8)$$

Unlike Equation 3, the cost in Equation 8 is independent of the location of e^- is the space. Also, this cost is always greater than or equal to the cost in Equation 3.

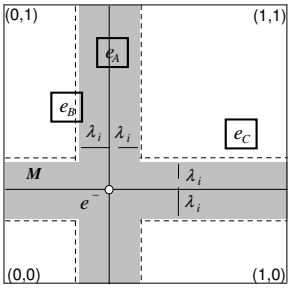
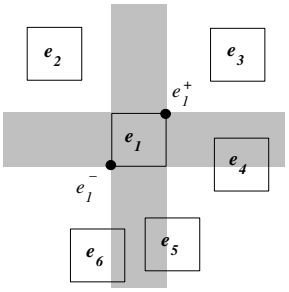


Fig. 9 I/O cost of computing upper bound

Lightweight upper bound score counting. In contrast to SCG, LCG utilizes a lightweight counting technique in order to obtain upper bound scores of non-leaf entries with low cost (see Section 4.2). We now present a modification of this

technique for deriving an upper bound score $\tau_r^u(e)$ for the entry e such that: (i) the computation requires no accesses to leaf nodes (thus saving significant cost), and (ii) $\tau_r^u(e)$ always upper bounds the exact $\tau_r(e^-)$. The access cost of this lightweight technique is given by Equation 8, except that leaf nodes (at level 0) are ignored.

Figures 10a,b exemplify how to obtain the value $\tau_r^u(e_1)$ for the entry e_1. The technique is the same as the one for computing the exact $\tau_r(e_1^-)$ value, except that level-1 entries (i.e., pointing to leaf nodes) are handled in another way. For the sake of demonstration, suppose that all entries shown in Figure 10a are level-1 entries. For any encountered level-1 entry (say, e_z), we increment $\tau_r^u(e_1)$ by $\psi^u(e_1, e_z) \cdot \mathtt{COUNT}(e_z)$, regardless of the $\psi^l(e_1, e_z)$ value.



(a) The case of an entry

Entry	Action on $\tau_r^u(e_1)$
e_1	add (2^2-1)COUNT(e_2)
e_2	add (2^1-1)COUNT(e_2)
e_3	add (2^2-1)COUNT(e_3)
e_4	add (2^2-1)COUNT(e_4)
e_5	add (2^1-1)COUNT(e_5)
e_6	add (2^1-1)COUNT(e_6)

(b) Derivation of $\tau_r^u(e_1)$

Fig. 10 Lightweight computation of the τ_r^u value of an entry

7.4 Adaptation of Priority-Based Traversal

In this section, we propose a priority-based traversal solution, called RelaxedPBT (Algorithm 8), for processing the relaxed query. The set S, the min-heap W, and the value γ have the same interpretation as in PBT (Algorithm 4). The major differences of RelaxedPBT from PBT are: (i) initialization of score bounds (Line 4–7), (ii) adjustment of score bounds (Lines 11–17), and (iii) elimination of unqualified entries (Line 20). As we will see later, several operations of the algorithm rely on the following property:

Property 1 Consider two (non-leaf) entries e and e' of the tree and a binding integer value α. If $\psi^l(e, e') = \psi^u(e, e') = \alpha$, then $\psi(p, p') = \alpha$ for any $p \in e$, $p' \in e'$.

Proof When $\psi^l(e, e')$ equals to $\psi^u(e, e')$, $\omega^l(e, e')$ is identical to $\omega^u(e, e')$. In this case, e and e' do not intersect along any dimension. Combining this fact with the bounding property of entries, we have $\omega(p, p') = \omega^l(e, e') = \omega^u(e, e')$, for any $p \in e$, $p' \in e'$. As a result, we obtain $\psi(p, p') = \alpha$.

RelaxedPBT begins by examining entries in the root node of the tree and deriving their lower/upper bound scores based on only entries in the root node. In each iteration (of the loop at Lines 8–20), a non-leaf entry e_z is selected from S according to a *priority order* (see Section 5.2). The child node (say, Z) of e_z is then read from the disk.

At Lines 11–13, we update score bounds for existing entries e_y in S, by comparing them against e_z. By Property 1, we need not adjust the score bounds of e_y when $\psi^u(e_y, e_z) = \psi^l(e_y, e_z)$. In case of $\psi^u(e_y, e_z) > \psi^l(e_y, e_z)$, the score contribution of e_z to e_y is replaced by those of entries in Z. Based on the same logic, Lines 14–17 are used to adjust score bounds of entries e_x in Z.

Next, we insert entries of Z into the set S and update the top-k results in W with entries (in S) having lower bound score $\tau_r^l(e)$ above γ. At Line 20, an entry e_m is removed from S when (i) its upper bound score $\tau_r^u(e_m)$ is below γ, and (ii) it cannot be used to adjust score bounds of any other entry in S with upper bound score above γ. The loop continues until S does not contain any no non-leaf entries. Finally, W is returned as the result.

Algorithm 8 Variant of PBT for the relaxed query

algorithm RelaxedPBT(Tree R, Integer k)
1: S:=new set; ▷ entry format in S: $\langle e, \tau_r^l(e), \tau_r^u(e)\rangle$
2: W:=new min-heap; ▷ k points with the highest τ_r^l
3: γ:=0; ▷ the k-th highest τ_r^l score found so far
4: **for all** $e_x \in R.root$ **do**
5: $\tau_r^l(e_x)$:=$\sum_{e \in R.root} \psi^l(e_x, e) \cdot \mathtt{COUNT}(e)$;
6: $\tau_r^u(e_x)$:=$\sum_{e \in R.root} \psi^u(e_x, e) \cdot \mathtt{COUNT}(e)$;
7: insert e_x into S and update W;
8: **while** S contains non-leaf entries **do**
9: remove e_z: non-leaf entry of S with the highest *priority*;
10: read the child node Z pointed by e_z;
11: **for all** $e_y \in S$ such that $\psi^u(e_y, e_z) > \psi^l(e_y, e_z)$ **do**
12: $\tau_r^l(e_y)$:=$\tau_r^l(e_y) - \psi^l(e_y, e_z) \cdot \mathtt{COUNT}(e_z) + \sum_{e \in Z} \psi^l(e_y, e) \cdot \mathtt{COUNT}(e)$;
13: $\tau_r^u(e_y)$:=$\tau_r^u(e_y) - \psi^u(e_y, e_z) \cdot \mathtt{COUNT}(e_z) + \sum_{e \in Z} \psi^u(e_y, e) \cdot \mathtt{COUNT}(e)$;
14: S_z:=$Z \cup \{\ e \in S \mid \psi^u(e_z, e) > \psi^l(e_z, e)\ \}$;
15: **for all** $e_x \in Z$ **do**
16: $\tau_r^l(e_x)$:=$\tau_r^l(e_z) - \psi^l(e_z, e_z) \cdot \mathtt{COUNT}(e_z) + \sum_{e \in S_z} \psi^l(e_x, e) \cdot \mathtt{COUNT}(e)$;
17: $\tau_r^u(e_x)$:=$\tau_r^u(e_z) - \psi^u(e_z, e_z) \cdot \mathtt{COUNT}(e_z) + \sum_{e \in S_z} \psi^l(e_x, e) \cdot \mathtt{COUNT}(e)$;
18: insert all entries of Z into S;
19: update W (and γ) by $e' \in S$ whose score bounds changed;
20: remove entries e_m from S where $\tau_r^u(e_m) < \gamma$ and $\neg\exists e \in S,\ (\tau_r^u(e) \geq \gamma) \wedge (\psi^u(e, e_m) > \psi^l(e, e_m))$;
21: **report** W as the result;

8 Experimental Evaluation

In this section, we experimentally evaluate the performance of the proposed algorithms. All algorithms in Table 2 were implemented in C++ and experiments were run on a Pentium D 2.8GHz PC with 1GB of RAM. For fairness to the STD algorithm [24], it is implemented with the spatial aggregation technique (discussed in Section 2.1) for optimizing counting

operations on aR-trees. In Section 8.1 we present an extensive experimental study for the efficiency of the algorithms with synthetically generated data. Section 8.2 studies the performance of the algorithms on real data and demonstrates the meaningfulness of top-k dominating points. Section 8.3 investigates the efficiency of our solutions for processing top-k dominating queries on non-indexed data. Section 8.4 presents the experimental study for the relaxed top-k dominating query.

Name	Description
STD	Skyline-Based Top-k Dominating Algorithm [24]
ITD	Optimized version of STD (Sec. 3.2)
SCG	Simple Counting Guided Algorithm (Sec. 4)
LCG	Lightweight Counting Guided Algorithm (Sec. 4)
UBT	Upper-bound Based Traversal Algorithm (Sec. 5)
CBT	Cost-Based Traversal Algorithm (Sec. 5)

Table 2 Description of the algorithms

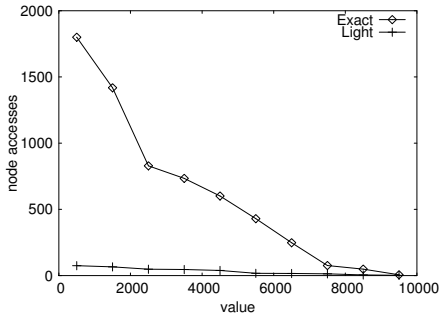
8.1 Experiments with Synthetic Data

Data generation and query parameter values. We produced three categories of synthetic datasets to model different scenarios, according to the methodology in [2]. UI contains datasets where point coordinates are random values *uniformly and independently generated* for different dimensions. CO contains datasets where point coordinates are *correlated*. In other words, for a point p, its i-th coordinate $p[i]$ is close to $p[j]$ in all other dimensions $j \neq i$. Finally, AC contains datasets where point coordinates are *anti-correlated*. In this case, points that are good in one dimension are bad in one or all other dimensions. Table 3 lists the range of parameter values and their default values (in bold type). Each dataset is indexed by an aR-tree with 4K bytes page size. We used an LRU memory buffer whose default size is set to 5% of the tree size.

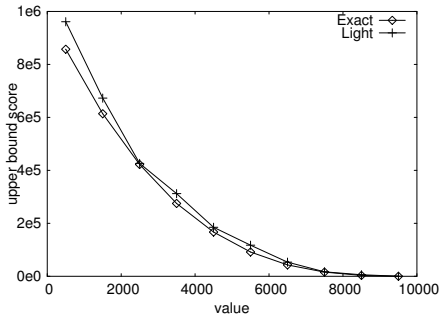
Parameter	Values
Buffer size (%)	1, 2, **5**, 10, 20
Data size, N (million)	0.25, 0.5, **1**, 2, 4
Data dimensionality, d	2, **3**, 4, 5
Number of results, k	1, 4, **16**, 64, 256

Table 3 Range of parameter values

Lightweight counting optimization in Counting-Guided search. In the first experiment, we investigate the performance savings when using the lightweight counting heuristic in the counting-guided algorithm presented in Section 4. Using a default uniform dataset, for different locations of a non-leaf entry e^-, (after fixing all coordinates of e^- to the same value v), we compare (i) node accesses of computing the exact $\tau(e^-)$ with that of computing a conservative upper bound $\tau^u(e)$ using the lightweight approach and (ii) the difference between these two bounds. Figure 11a shows the effect of v (i.e., location of e^-) on node accesses of these two computations. Clearly, the lightweight approach is much more efficient than the exact approach. Their cost difference can be two orders of magnitude when e^- is close to the origin. Figure 11b plots the effect of v on the value of upper bound score. Even though lightweight computation accesses much fewer nodes, it derives a score that tightly upper bounds the exact score ($\tau^u(e)$ is only 10% looser than $\tau(e^-)$). Summarizing, the lightweight approach is much more efficient than the exact approach while still deriving a reasonably tight upper bound score.



(a) Node accesses

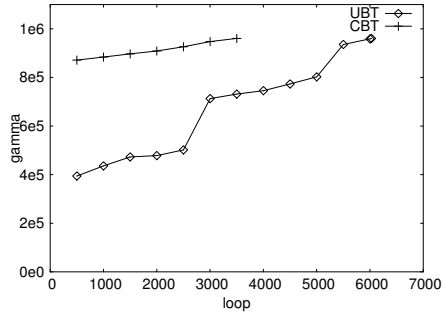


(b) Upper bound score

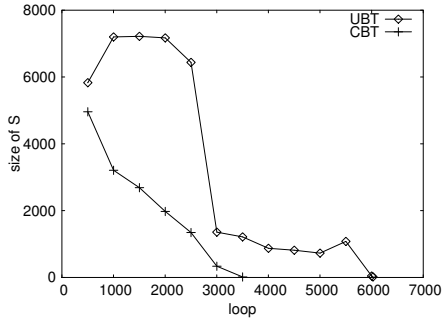
Fig. 11 The effect of v, UI, N =1M, $d = 3$

Orderings in Priority-Based Traversal. In Section 5.2, we introduced two priority orders for selecting the next non-leaf entry to process at PBT: (i) UBT chooses the one with the highest upper bound score, and (ii) CBT, among those with the highest level, chooses the one with the highest upper bound score. Having theoretically justified the superiority of CBT over UBT (in Section 5.2), we now demonstrate this experimentally. For the default top-k dominating query on a UI dataset, we record statistics of the two algorithms during their execution. Figure 12a shows the value of γ (i.e., the best-k score) for both UBT and CBT as the number of loops executed. Note that in UBT/CBT, each loop (i.e., Lines 8–20 of Algorithm 4) causes one tree node access. Since γ rises faster in CBT than in UBT, CBT has higher pruning power and thus terminates earlier. Figure 12b plots the size of S (i.e., number of entries in memory) with respect to the number of loops. The size of S in CBT is much lower than that

in UBT. Hence, CBT requires less CPU time than UBT on book-keeping the information of visited entries and negligible memory compared to the problem size. Both figures show that our carefully-designed priority order in CBT outperforms the intuitive priority order in UBT by a wide margin.



(a) Value of γ



(b) Size of S

Fig. 12 The effect of ordering priorities, UI, N =1M, $d = 3$

Comparison of all algorithms and variants thereof. We now compare all algorithms and their variants (STD, ITD, SCG, LCG, UBT, CBT) for the default query parameters on UI, CO, and AC datasets (Figure 13). In this and subsequent experiments, we compile the I/O and CPU costs of each algorithm, by charging 10ms I/O time per page fault, and show their I/O-CPU cost-breakdown. ITD performs much better than the baseline STD algorithm of [24] (even though STD operates on the aR-tree), due to the effectiveness of the batch counting and Hilbert ordering techniques for retrieved (constrained) skyline points. LCG and CBT significantly outperform ITD, as they need not compute the scores for the whole skyline, whose size grows huge for AC data. Note that the optimized version of counting-guided search (LCG) outperforms the simple version of the algorithm that computes exact upper bounds (SCG) by a wide margin. Similarly, for priority-based traversal, CBT outperforms UBT because of the reasons explained in the previous experiment. Observe that the best priority-traversal algorithm (CBT) has lower I/O cost than optimized counting-guided search (LCG), since CBT accesses each node at most once but LCG may access some nodes more than once during counting operations.

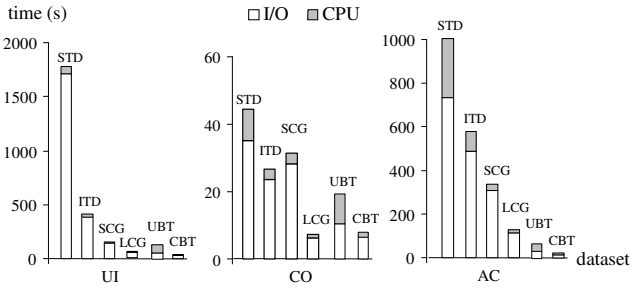
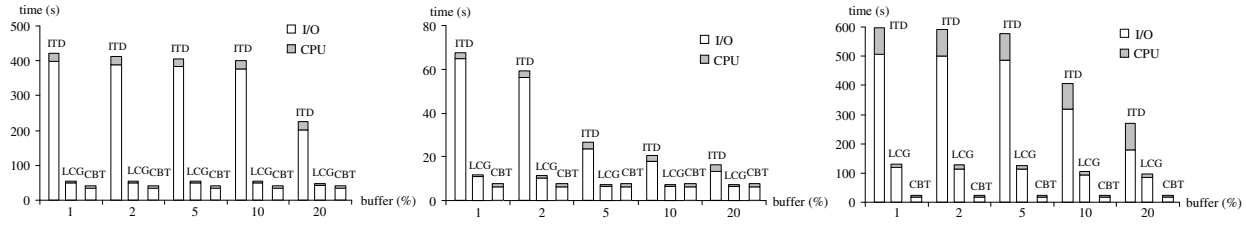


Fig. 13 Query cost ($k = 16$, N =1M, $d = 3$)

In remaining experiments, we only compare the best algorithms from each gender (ITD, LCG, and CBT), for a wide range of query and system parameter values. First, we study the effect of the buffer on the performance of the algorithms. Figure 14 shows the cost of the algorithms as a function of buffer size (%). Observe that the costs of LCG and CBT with the smallest tested buffer (1% of the tree size) are still much lower than that of ITD with the largest buffer size (20%). Since CBT accesses each tree node at most once, its cost is independent of the buffer. Clearly, CBT outperforms its competitors for all tested buffer sizes. We note that the memory usage (for storing visited tree entries) of ITD, LCG, and CBT for UI data are 0.03%, 0.02%, 0.96% of the tree size, respectively, and are further reduced by 30% for CO data. For AC data the corresponding values are 2.72%, 0.11%, and 1.48%. Besides, their memory usage increases slowly with k and rises sublinearly with N. Even at $d = 5$, their memory usage is only two times of that at $d = 3$.

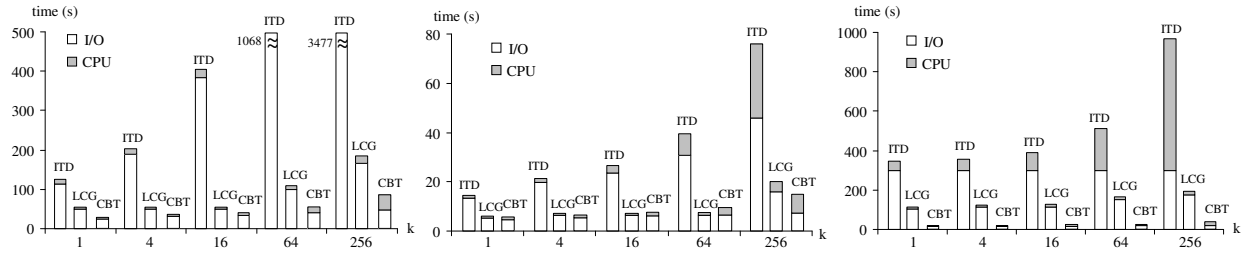
We also investigated the effect of k on the cost of the algorithms (see Figure 15). In some tested cases of Figure 15a, the cost of ITD is too high for the corresponding bar to fit in the diagram; in these cases the bar is marked with a "$\approx$" sign and the actual cost is explicitly given. Observe that LCG and CBT outperform ITD in all cases. As k increases, ITD performs more constrained skyline queries, leading to more counting operations on retrieved points. CBT has lower cost than LCG for UI data because CBT accesses each tree node at most once. For CO data, counting operations in LCG become very efficient and thus LCG and CBT have similar costs. On the other hand, for AC data, there is a wide performance gap between LCG and CBT.

Figure 16 plots the cost of the algorithms as a function of the data dimensionality d. Again, ITD is inferior to its competitors for most of the cases. As d increases, the number of skyline points increases rapidly but the number of points examined by LCG/CBT increases at a slower rate. Again, CBT has lower cost than LCG for all cases. Figure 17 investigates the effect of the data size N on the cost of the algorithms. When N increases, the number of skyline points increases considerably and ITD performs much more batch counting



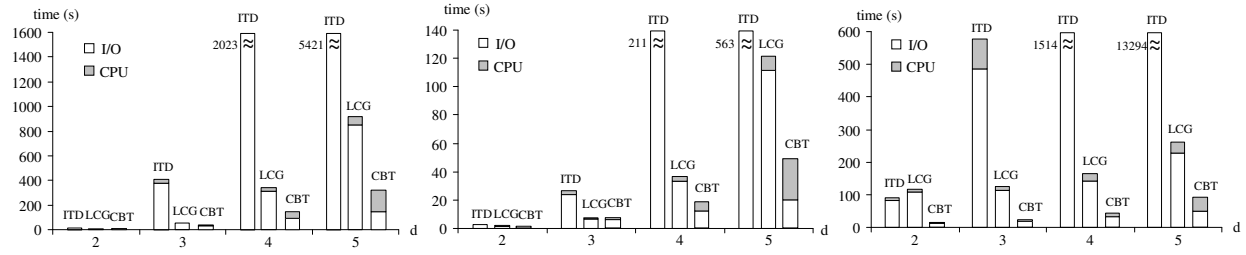
(a) UI (b) CO (c) AC

Fig. 14 Cost vs. buffer size (%), $k = 16$, N =1M, $d = 3$



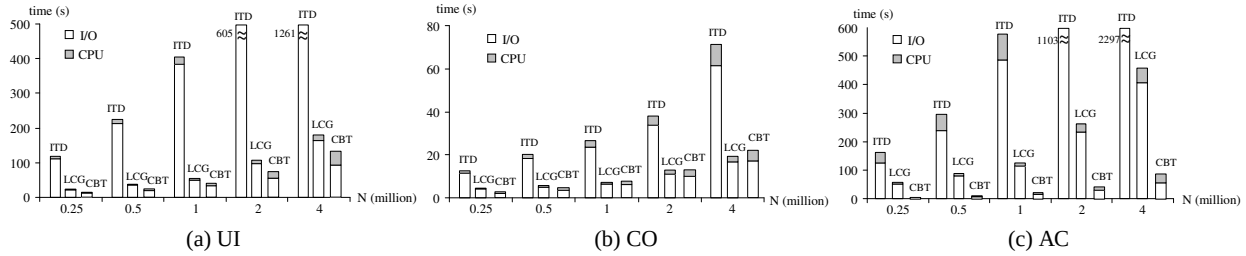
(a) UI (b) CO (c) AC

Fig. 15 Cost vs. k, N =1M, $d = 3$



(a) UI (b) CO (c) AC

Fig. 16 Cost vs. d, N =1M, $k = 16$



(a) UI (b) CO (c) AC

Fig. 17 Cost vs. N, $d = 3$, $k = 16$

operations than LCG. Also, the performance gap between LCG and CBT widens.

8.2 Experiments with Real Data

Datasets. We experimented with three real multi-dimensional datasets: *FC*[5], *NBA*[6], and *BASEBALL*[7]. *FC* contains 581012 forest land cells (i.e., data objects), having four attributes: horizontal distance to hydrology (*hh*), vertical distance to hydrology (*vh*), horizontal distance to roadways (*hr*), and horizontal distance to fire points (*hf*). For *FC*, small values are preferable to large ones at all dimensions. *NBA* contains regular season statistics of 19112 NBA players (i.e., data objects). In order for the query to be meaningful, only few important attributes are selected for NBA players: games played (*gp*), points (*pts*), rebounds (*reb*), and assists (*ast*). *BASEBALL* consists of statistics of 36898 baseball pitchers (i.e., data objects). Similarly, few important attributes are chosen for baseball pitchers: wins (*w*), games (*g*), saves (*sv*), and strikeouts (*so*). In the last two datasets, large values are preferable for all dimensions and each player is uniquely identified by his/her name and year.

Performance Experiment. Table 4 shows the cost of the algorithms on two largest datasets (*FC* and *BASEBALL*) for

[5] Forest cover dataset, UCI KDD Archive. http://kdd.ics.uci.edu
[6] NBA Statistics v2.0. http://basketballreference.com
[7] The Baseball Archive v5.3. http://baseball1.com/statistics

different values of k, by fixing the buffer size to 5% of the tree size. Observe that the cost of ITD becomes prohibitively expensive at high values of k. Clearly, CBT has the lowest cost and the performance gap between the algorithms widens as k increases.

	time (seconds)					
	FC			*BASEBALL*		
k	**ITD**	**LCG**	**CBT**	**ITD**	**LCG**	**CBT**
1	262.3	162.0	62.0	4.6	13.0	0.9
4	413.0	166.6	69.7	9.4	16.5	1.8
16	814.2	204.2	78.9	22.8	18.4	2.5
64	2772.7	282.2	99.4	69.7	22.8	3.5
256	9942.1	523.0	176.4	271.1	38.6	5.9

Table 4 Query cost vs. k, real datasets

Meaningfulness of top-k dominating query results. Table 5 shows the dominating scores and the attribute values of the top-5 dominating players in the *NBA* and *BASEBALL* datasets. Readers familiar with these sports can easily verify that the returned results match the public view of super-star players. Although the ranking of objects by their τ-scores may not completely match with every personalized ranking suggested by individuals, a top-k dominating query at least enables them to discover some representative "top" objects without any specific domain knowledge. In addition, we note that some of the top-k results do not belong to the skyline. For example, the *NBA* player "Kevin Garnett / 2002" is the top-3 result, even though he is dominated by the top-1 result (i.e., not a skyline point). Similarly, the top-4 *BASEBALL* pitcher is dominated by the top-2. These players could not be identified by skyline queries.

Score	**NBA Player / Year**	**gp**	**pts**	**reb**	**ast**
18585	Wilt Chamberlain / 1967	82	1992	1952	702
18299	Billy Cunningham / 1972	84	2028	1012	530
18062	Kevin Garnett / 2002	82	1883	1102	495
18060	Julius Erving / 1974	84	2343	914	462
17991	Kareem Abdul-Jabbar / 1975	82	2275	1383	413

Score	**BASEBALL Pitcher / Year**	**w**	**g**	**sv**	**so**
34659	Ed Walsh / 1912	27	62	10	254
34378	Ed Walsh / 1908	40	66	6	269
34132	Dick Radatz / 1964	16	79	29	181
33603	Christy Mathewson / 1908	37	56	5	259
33426	Lefty Grove / 1930	28	50	9	209

Table 5 Top-5 dominating players

In general, various approaches could be applied to measure the *meaningfulness* of query results. Yet, there is no standardized notion for capturing the meaningfulness of results. We regard the τ score as a reasonable, obvious, and quantitative measure of the result meaningfulness; due to the rationale that, each individual top-k dominating player is guaranteed to overqualify a large number of other players (in other teams). However, we are not advocating the τ score as the best possible measure of result meaningfulness.

As an alternative choice of result meaningfulness, we also measure the number of distinct data points dominated by the query result set [22], on real datasets. For the *NBA* dataset, the top-1, top-2, and top-5 (dominating) query results dominate respectively 97.24%, 98.13%, and 98.80% of distinct points in the dataset. For the *BASEBALL* dataset, the top-1, top-2, and top-5 (dominating) query results dominate respectively 93.93%, 94.88%, and 98.67% of data points. It turns out that, some points in the result set are well separated from the others, causing the overall result set to dominate a substantial number of distinct data points.

8.3 Experiments with Non-indexed Data

In this section, we evaluate the performance of our proposed solutions for top-k dominating queries on non-indexed data. We use CRS to denote the version of our algorithm with uses the CRS-filter in the filter pass. The version using the FN-Filter has variants with different search orderings in the filter step: (i) FNS, with the sweep-line ordering, (ii) FNU, with the upper bound score ordering, and (iii) FNP, with the partial-dominance reduction ordering. As a reference, we compare these methods with CBT, which is the best aR-tree based algorithm. In order to apply CBT, we need to bulk-load the aR-tree from the data first, so we include the cost of the tree creation in its overhead.

Note that the I/O accesses of our non-indexed solutions (and the bulk-loading stage before CBT) are mostly sequential (with negligible random disk page accesses). Each sequential page access is charged 1ms I/O time. For instance, CRS performs three full read passes over data. Each fine-grained solution (i.e., FNS, FNU, FNP) performs one full read pass and one full write pass in the in the counting pass, and two partial read passes (i.e., some partitions are not accessed in filter and refinement steps). For fairness to CBT, we assume that the main memory is large enough for the aR-tree bulk-loading stage to complete in two full read passes and two full write passes.

Figure 18 illustrates the cost breakdown of our proposed methods on non-indexed data, for default parameter values on UI, CO, and AC datasets. Each bar is decomposed into filter CPU time, refinement CPU time, and the total sequential I/O time (of all steps/passes). For CBT, sequential I/O time indicates its cost in the bulk-loading stage, whereas its filter time represents the total query evaluation time (i.e., CPU time and random I/O time) using the aR-tree. Due to the bulk-loading stage, CBT is more expensive than most of our non-indexed methods, especially for the UI dataset. CRS is a coarse-grained solution so its filter step is cheap; however, many candidates are produced and the refinement step is expensive. In particular, its computational time is high for the AC dataset, because of the huge candidate size. On the other hand, the fine-grained solutions (FNS, FNU,

FNP) have robust performance across different data distributions because they tighten score bounds of existing candidates while reading new points in the filter step.

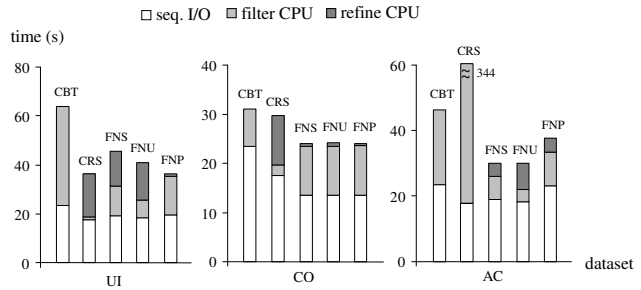


Fig. 18 Query cost on non-indexed data ($k = 16$, N =1M, $d = 3$)

We proceed to examine the filter effectiveness of the proposed non-indexed solutions. Specifically, we measure the candidate size $|C|$ and the top-k lower bound score γ (known so far) at the end of the filter step. Both of them provide the user early insight about the results. Table 6 shows the values of $|C|$ and γ, obtained by our methods, on different data distributions. As a comparison, we include into the last row the number of results and the actual top-k score. In summary, FNP has the best filter effectiveness, followed by FNS, FNU, and CRS. Since CRS relies mainly on the dominance property to prune unqualified points, it can hardly reduce the candidate size for the AC dataset. FNU is a fine-grained solution and performs tightening of score bounds for candidate points in the filter step; thus, it is more effective than CRS. However, FNU visits the disk partitions in descending order of their upper bound scores, and it shares the same drawback as its tree-based counterpart UBT (see Section 8.1). Interestingly, the visiting order of FNS is independent of the underlying data distributions, yet it is more effective than FNU. The FNP method, with our carefully-designed visiting order, leads to extremely low candidate sizes $|C|$ and tight top-k lower bound score γ. In particular, for the UI data, the candidate set of FNP is exactly the same as the final result set and γ is only 0.002% lower than the actual top-k score. Therefore, we recommend FNP as the best non-indexed solution for top-k dominating queries.

Method	UI		CO		AC	
	$\|C\|$	γ	$\|C\|$	γ	$\|C\|$	γ
CRS	616	669651	522	841191	34575	10773
FNS	411	821608	125	991319	135	91530
FNU	466	762140	154	990137	2864	89452
FNP	16	960650	93	992488	48	123315
Results	16	960670	16	994637	16	123462

Table 6 Candidate size $|C|$ and top-k score γ, ($k = 16$, N =1M, $d = 3$)

We then investigate the progressiveness of our non-indexed solutions. During the execution of an algorithm, the top-k lower bound score γ (known so far) provides the user an early and rough picture over the actual score. Figure 19 plots the γ value of the algorithms (CBT, FNS, FNP) as a function of time (including both I/O time and CPU time). Observe that both FNS and FNP acquire high γ value early at 10–15s. Since the application of CBT on non-indexed data requires aR-tree bulk-loading, it starts obtaining high γ value only after 25s. In summary, both FNS and FNP allow the user to attain early a tight lower bound estimate of the actual top-k score.

8.4 Experiments with the Relaxed Query

Performance Experiment. Figure 20 shows the cost of our algorithms for the relaxed top-k dominating query on UI, CO, and AC datasets, with the default parameter values. In general, CBT has the best performance and it is stable for different data distributions. Since ITD and LCG access some tree node multiple times (through different counting operations), they become expensive for processing the relaxed query, especially on the AC dataset. In contrast, CBT reads each tree node at most once and adjusts score bounds of existing entries incrementally.

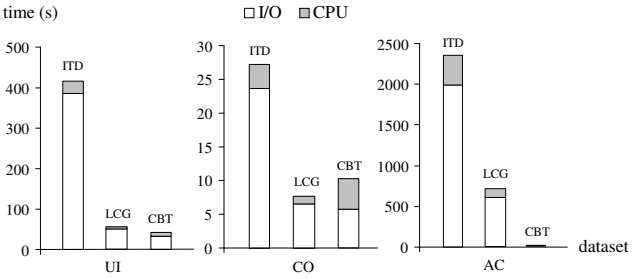
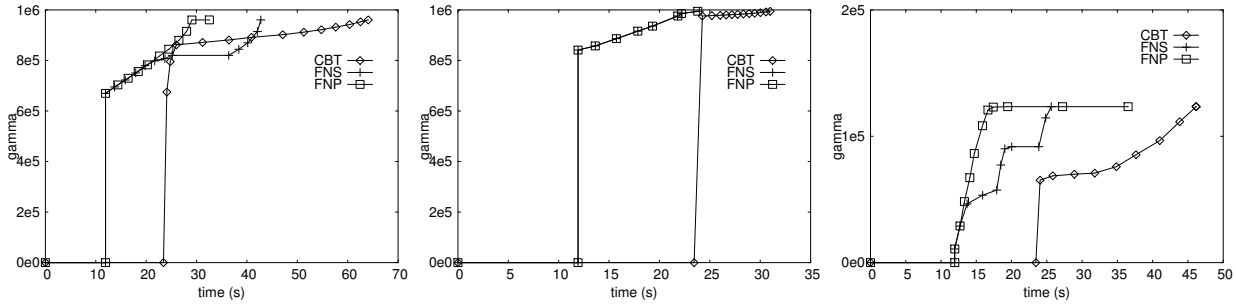


Fig. 20 Query cost of relaxed query ($k = 16$, N =1M, $d = 3$)

Data Analysis on Real Data with Missing Values. In real-life, the data may have missing values, either inherently, or introduced by the data owner in purpose. This may happen, for example, in an attempt to avoid leakage of sensitive values. Another example is that the data owner chooses to publish a "trial" dataset with missing values and only reveals the original dataset to the client upon purchase.

We now demonstrate the robustness of the relaxed query on a real dataset with missing values. Specially, for each tuple in the *NBA* dataset, an attribute is randomly chosen and its value is set to `NULL`. The resulting dataset is called the *NBA*$_{miss}$ dataset. Since our algorithms operate on aR-tree indexed data, each `NULL` value in the tree needs to be replaced by the worst value. Table 7 shows the relaxed top-16 dominating players on the *NBA*$_{miss}$ dataset. The results are then compared with the top-70 dominating points on the original dataset *NBA*. For instance, the 4-th point in *NBA*$_{miss}$ is the 7-th point in *NBA*; the 5-th point in *NBA*$_{miss}$ is marked as "—", meaning that it is outside the top-70 in *NBA*. It turns



(a) UI (b) CO (c) AC

Fig. 19 Top-k score γ vs. time, N =1M, $k = 16$

out that, the relaxed query is able to retrieve a decent number of meaningful results, even though in the presence of many missing values in *NBA*$_{miss}$. The robustness of the relaxed query is explained by the fact that the contribution of a score component is less restrictive in the (relaxed) τ_r function than in the (original) τ function.

τ_r rank on *NBA*$_{miss}$	original rank on *NBA*	NBA Player / Year
1	2	Billy Cunningham / 1972
2	—	—
3	4	Julius Erving / 1974
4	7	Kevin Garnett / 2004
5	—	—
6	6	Don Adams / 1975
7	44	John Havlicek / 1970
8	—	—
9	9	Julius Erving / 1973
10	—	—
11	48	Rogera Brown / 1969
12	—	—
13	12	Larry Bird / 1980
14	62	Billy Cunningham / 1969
15	—	—
16	59	Kevin Garnett / 2001

Table 7 Relaxed top-16 dominating players on the *NBA*$_{miss}$ dataset

9 Discussion

In this section, we present the summary of our experimental results and discuss the scalability of the proposed techniques for high dimensional data.

Summary of Experimental Results. Regarding the processing of top-k dominating queries on aR-tree indexed data, our performance experiments suggest that CBT has stable performance across different data distributions. Also, it has the best performance for the case of relaxed top-k dominating queries. Thus, it is recommended for evaluating top-k dominating queries on indexed data.

In case the data is not indexed, the FNP method outperforms its competitors and it has robust performance for different data distributions. In addition, its processing cost is better than the best index-based approach (CBT), if the latter includes the cost of bulk-loading the index.

High Dimensional Data. Recall that, in Equation 1, the score of point p is defined by the number of points p' dominated by p. When the dimensionality of the problem is high, the dominance condition becomes too restrictive and even the top points may have low scores. Consequently, there may not exist a distinctive top object having much higher scores than the rest, implying that the top-k dominating query is not meaningful, due to the dimensionality curse. In order to produce meaningful results, we consider only low dimension data (from 2 to 5) in our experiments. In addition, both our indexed and non-indexed algorithms become inefficient for high dimensional data.

To extend the applicability of top-k dominating analysis for high dimensional data, we introduce the relaxed top-k dominating query, which is able to capture "partial" dominance relationships among the data points. Thus, meaningful top-k results can be obtained from the relaxed query over high dimensional data. Still, our techniques proposed in Section 7 operate on multi-dimensional indexes or grids, which degenerate at high dimensionality. As part of our future work, we will focus on the development of efficient solutions for the relaxed query over high dimensional data.

10 Conclusion

In this paper, we studied the interesting and important problem of processing top-k dominating queries on multi-dimensional data. Although the skyline-based algorithm in [24] is applicable to the problem, it suffers from poor performance, as it unnecessarily examines many skyline points. This motivated us to develop carefully-designed solutions that exploit the intrinsic properties of the problem for accelerating query evaluation. First, we proposed ITD, which integrates the algorithm of [24] with our optimization techniques (batch counting and Hilbert ordering). Next, we developed LCG, a top-k dominating algorithm that guides search by computing upper bound scores for non-leaf entries, and utilizes a lightweight (i.e., I/O-inexpensive) tech-

nique for computing upper bound scores. Then, we proposed I/O efficient algorithm CBT that accesses each node at most once. The effectiveness of our optimizations (lightweight counting technique in LCG and traversal order in CBT) were analyzed theoretically.

In addition to algorithms that apply on indexed data, we also propose a methodology for evaluating top-k dominating queries over non-indexed data that are stored in a sequential file. Our method can compute the query result within three passes over the data. In the first pass, a grid-histogram is computed to capture the distribution of the points. The grid is used to derive three types of bounds for multi-dimensional regions, which are helpful to determine a set of candidate top-k points during the second pass. In the third and final pass, the dominance scores of the candidates are counted exactly to derive the final result. We proposed and compared variants for the second (filter) pass of the algorithm.

The final contribution of the paper is the proposal of a relaxed version of the top-k dominating query, where the dominance relationships between points in all dimensional subspaces are considered. The score of a point is determined by summing the number of points it dominates from all subspaces. We exemplified and showed experimentally the flexibility of this query compared to the strict version of the problem. In addition, we showed how the proposed algorithms can be adapted to solve this relaxed top-k dominating query.

Acknowledgement

This work was supported by grant HKU 7149/07E from Hong Kong RGC.

References

1. W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient Distributed Skylining for Web Information Systems. In *EDBT*, 2004.
2. S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, 2001.
3. A. R. Butz. Alternative Algorithm for Hilbert's Space-Filling Curve. *IEEE Trans. Comput.*, C-20(4):424–426, 1971.
4. C.-Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified Computation of Skylines with Partially-Ordered Domains. In *SIGMOD*, 2005.
5. C.-Y. Chan, H. Jagadish, K.-L. Tan, A. Tung, and Z. Zhang. Finding k-Dominant Skylines in High Dimensional Space. In *SIGMOD*, 2006.
6. C.-Y. Chan, H. Jagadish, K.-L. Tan, A. Tung, and Z. Zhang. On High Dimensional Skylines. In *EDBT*, 2006.
7. S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust Cardinality and Cost Estimation for Skyline Operator. In *ICDE*, 2006.
8. J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In *ICDE*, 2003.
9. R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.
10. P. Godfrey. Skyline Cardinality for Relational Processing. In *FoIKS*, 2004.
11. P. Godfrey, R. Shipley, and J. Gryz. Maximal Vector Computation in Large Data Sets. In *VLDB*, 2005.
12. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.
13. G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *TODS*, 24(2):265–318, 1999.
14. V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A System for the Efficient Execution of Multiparametric Ranked Queries. In *SIGMOD*, 2001.
15. Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline Queries Against Mobile Lightweight Devices in MANETs. In *ICDE*, 2006.
16. D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*, 2002.
17. I. Lazaridis and S. Mehrotra. Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure. In *SIGMOD*, 2001.
18. S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*, 1997.
19. C. Li, K. C.-C. Chang, and I. F. Ilyas. Supporting Ad-hoc Ranking Aggregates. In *SIGMOD*, 2006.
20. C. Li, B. C. Ooi, A. Tung, and S. Wang. DADA: A Data Cube for Dominant Relationship Analysis. In *SIGMOD*, 2006.
21. X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *ICDE*, 2005.
22. X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting Stars: The k Most Representative Skyline Operator. In *ICDE*, 2007.
23. D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP Operations in Spatial Data Warehouses. In *SSTD*, 2001.
24. D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline Computation in Database Systems. *TODS*, 30(1):41–82, 2005.
25. J. Pei, A. W.-C. Fu, X. Lin, and H. Wang. Computing Compressed Multidimensional Skyline Cubes Efficiently. In *ICDE*, 2007.
26. J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces. In *VLDB*, 2005.
27. J. Pei, Y. Yuan, X. Lin, W. Jin, M. Ester, Q. Liu, W. Wang, Y. Tao, J. X. Yu, and Q. Zhang. Towards Multidimensional Subspace Skyline Analysis. *TODS*, 31(4):1335–1381, 2006.
28. K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *VLDB*, 2001.
29. Y. Tao, X. Xiao, and J. Pei. SUBSKY: Efficient Computation of Skylines in Subspaces. In *ICDE*, 2006.
30. Y. Theodoridis and T. K. Sellis. A Model for the Prediction of R-tree Performance. In *PODS*, 1996.
31. M. L. Yiu and N. Mamoulis. Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data. In *VLDB*, 2007.
32. Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient Computation of the Skyline Cube. In *VLDB*, 2005.