Information Sciences 179 (2009) 3286–3308

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

The partitioned-layer index: Answering monotone top-*k* queries using the convex skyline and partitioning-merging technique

Jun-Seok Heo [a], Kyu-Young Whang [a,*], Min-Soo Kim [a], Yi-Reun Kim [a], Il-Yeol Song [b]

[a] Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, South Korea
[b] College of Information Science and Technology, Drexel University, Philadelphia, PA 19104, USA

ARTICLE INFO

Article history:
Received 28 July 2008
Received in revised form 26 May 2009
Accepted 27 May 2009

Keywords:
Top-*k* queries
Monotone linear score functions
Partitioning
Merging
Skylines
Convex hulls

ABSTRACT

A top-*k* query returns *k* tuples with the highest (or the lowest) scores from a relation. The score is computed by combining the values of one or more attributes. We focus on top-*k* queries having monotone linear score functions. Layer-based methods are well-known techniques for top-*k* query processing. These methods construct a database as a single list of layers. Here, the *i*th layer has the tuples that can be the top-*i* tuple. Thus, these methods answer top-*k* queries by reading at most *k* layers. Query performance, however, is poor when the number of tuples in each layer (simply, the *layer size*) is large. In this paper, we propose a new layer-ordering method, called the *Partitioned-Layer Index* (simply, the *PL Index*), that significantly improves query performance by reducing the layer size. The PL Index uses the notion of *partitioning*, which constructs a database as multiple sublayer lists instead of a single layer list subsequently reducing the layer size. The PL Index also uses the *convex skyline*, which is a subset of the skyline, to construct a sublayer to further reduce the layer size. The PL Index has the following desired properties. The query performance of the PL Index is quite insensitive to the weights of attributes (called the *preference vector*) of the score function and is approximately linear in the value of *k*. The PL Index is capable of tuning query performance for the most frequently used value of *k* by controlling the number of sublayer lists. Experimental results using synthetic and real data sets show that the query performance of the PL Index significantly outperforms existing methods except for small values of *k* (say, $k \leqslant 9$).

1. Introduction

A *top-k* (*ranked*) query returns *k* tuples with the highest (or the lowest) scores in a relation [18]. A score function is generally in the form of a linearly weighted sum as shown in Eq. (1) [10,15,22]. Here, $p[i]$ and $t[i]$ denote the weight and the value of the *i*th attribute of a tuple *t*, respectively. The *d*-dimensional vector that has $p[i]$ as the *i*th element is called the *preference vector* [15], where *d* denotes the number of attributes of *t*. In this paper, we focus on monotone linear score functions where $p[i] \geqslant 0 (1 \leqslant i \leqslant d)$. In Section 3, we explain the score function in detail.

$$f(t) = \sum_{i=1}^{d} p[i] * t[i] \tag{1}$$

* Corresponding author.
E-mail addresses: jsheo@mozart.kaist.ac.kr (J.-S. Heo), kywhang@mozart.kaist.ac.kr (K.-Y. Whang), mskim@mozart.kaist.ac.kr (M.-S. Kim), yrkim@mozart.kaist.ac.kr (Y.-R. Kim), songiy@drexel.edu (I.-Y. Song).

0020-0255/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2009.05.016

For example, top-k queries are used for ranking colleges. Colleges are represented by a relation that has numerical attributes such as the research quality assessment, tuition, and graduate employment rate [8,22]. Users search for the k colleges ranked according to the score function with their own preference vector [8,15,22]. Thus, users having budget concern may assign a high weight to the "tuition" attribute while users expecting good employment will assign a high weight to the "graduate employment rate" attribute. For another example, top-k queries are used for ordering houses listed for sale. Houses are represented by a relation that has numerical attributes such as the price, number of bedrooms, age, and square footage [15,22]. As shown in the previous example, users can search for the best houses with their own preference vector [8,15,22].

To process a top-k query, a naive method would calculate the score of each tuple according to the score function, and then, finds the top k tuples by sorting the tuples based on their scores. This method, however, is not appropriate for a query with a relatively small value of k over large databases because it incurs a significant overhead by reading even those tuples that cannot possibly be the results [22].

There have been a number of methods proposed to efficiently answer top-k queries by accessing only a subset of the database instead of unnecessarily accessing the entire one. These methods are classified into two categories depending on whether or not they exploit the relationship among the attributes (i.e., the *attribute correlation* [8]). The ones that do not exploit attribute correlation regard the attributes independent of one another. That is, they consider a tuple as a potential top-k answer only if the tuple is ranked high in *at least* one of the attributes. We refer to these methods as *list-based methods* because they require maintaining one sorted list per each attribute [2,11]. While these methods show significant improvement compared to the naive method, they often consider an unnecessarily large number of tuples. For instance, when a tuple is ranked high in *one* attribute but low in all others, the tuple is likely to be ranked low in the final answer and can potentially be ignored, but these methods have to consider it because of its high rank in that one attribute.

The ones that exploit attribute correlation regard the attributes dependent of one another. That is, they consider *all* attribute values when constructing an index in contrast to the list-based methods. These methods are further classified into two categories – layer-based methods and view-based methods [22]. The *view-based methods* build multiple dedicated indices for multiple top-k queries with different preference vectors. That is, these methods create these top-k queries, execute each query, and store the result as a view. These methods answer top-k queries by reading tuples from the view(s) whose preference vector is the most similar to that of a given query. These methods have a disadvantage of being sensitive to the preference vector [22]. If the preference vector used in creating the view is similar to that of a given query, query performance is good; otherwise, it is very poor [22]. Query performance can be improved by increasing the number of views, but the space overhead increases in proportion to the number of views [21].

The *layer-based methods* construct the database as a *single* list of layers (simply, a *layer list*) where the ith layer contains the tuples that can potentially be the top-i answer. These methods answer top-k queries by reading at most k layers from the layer list. For constructing layers, *convex hulls* [4] or *skylines*[1] [6] can be used [8,22]. The convex hull is a useful notion when supporting (monotone or non-monotone) *linear* functions [8]; the skyline (linear or non-linear) *monotone* functions [6]. The layer-based methods have two advantages over the view-based methods: 1) storage overhead is negligible [8], and 2) query performance is not very sensitive to the preference vector of the score function given by the query [22]. Nevertheless, when the number of tuples in each layer (simply, the *layer size*) is large, these methods have bad query performance because many unnecessary tuples have to be read to process the query [15].

In this paper, we propose a new layer-based method, called the *Partitioned-Layer Index* (simply, the *PL Index*), that significantly improves top-k query processing by reducing the layer size. The PL Index overcomes the drawback of the layer-based methods. Besides, since the PL Index belongs to a layer-based approach, it inherently takes advantage of exploiting attribute correlation in contrast to the list-based methods and has the two desirable properties over the view-based methods. The contributions we make in this paper are as follows.

(1) We propose the notion of *partitioning* for constructing multiple sublayer lists instead of a single layer list. We partition the database into a number of *distinct* parts, and then, constructs a sublayer lists for each part. That is, the PL Index consists of a set of sublayer lists. The partitioning method allows us to reduce the sizes of the sublayers inversely proportional to the number of sublayer lists. Accordingly, the PL Index overcomes the drawback of existing layer-based methods of large layer sizes while avoiding the space overhead of the view-based methods.
(2) By using the partitioning method, we propose the novel PL Index and its associated top-k query processing algorithm. Our algorithm dynamically constructs a virtual layer list by merging the sublayer lists of the PL Index to process a specific top-k query and returns the query results progressively. A novel feature of our algorithm is that it finds a top-i ($2 \leqslant i \leqslant k$) tuple by reading *at most one sublayer* from a specific sublayer list.
(3) We formally define the notion of the *convex skyline*, which is a subset of the skyline, to further reduce the sublayer size. The convex skyline is a notion more useful than either the skyline or the convex hull when supporting monotone linear score functions due to this characteristic of reducing the layer size.

The PL Index has four desirable properties. First, it significantly outperforms existing methods except for small values of k (say, $k \leqslant 9$). Second, its query performance is quite insensitive to the preference vector of the score function. Third, it is capa-

[1] The definition of the skyline and its property will be presented in Section 4.3.

ble of tuning the query performance for the most frequently used value of k by controlling the number of sublayer lists. Fourth, its query performance is approximately linear in the value of k. We investigate these properties in Section 6 and present the results of performance evaluation in Section 7.

The rest of this paper is organized as follows. Section 2 describes existing work related to this paper. Section 3 formally defines the problem. Section 4 proposes the method for building the PL Index. Section 5 presents the algorithm for processing top-k queries using the PL Index. Section 6 analyzes the performance of the PL Index in index building and querying. Section 7 presents the results of performance evaluation. Section 8 summarizes and concludes the paper.

2. Related work

There have been a number of methods proposed to answer top-k queries efficiently by accessing only a subset of the database. We classify the existing methods into three categories: the *list-based method*, *layer-based method*, and *view-based method*. We briefly review each of these methods in this section.

2.1. Layer-based method

The layer-based method constructs a global index based on the combination of *all* attribute values of each tuple. Within the index, tuples are partitioned into multiple layers, where the ith layer contains the tuples that can potentially be the top-i answer. Therefore, the top-k answers can be computed by reading at most k layers. We explain in detail ONION [8] and AppRI [22], which are the layer-based methods that are closely related to this paper.

2.1.1. ONION

ONION [8] builds the index by making layers with the vertices (or the *extreme points* [13]) of the *convex hulls* [4] over the set of tuples represented as point objects in the multi-dimensional space. That is, it makes the first layer with the convex hull vertices over the entire set of tuples, and then, makes the second layer with the convex hull vertices over the set of remaining tuples, and so on. As a result, an outer layer geometrically encloses inner layers. By using the concept of optimally linearly ordered set in Definition 1 below, Chang et al. [8] have proved that ONION answers top-k queries by reading at most k layers starting from the outmost layer. In Definition 1, for finding k tuples with the *lowest scores* as the results, we flip the inequality sign between $p \cdot x$ and $p \cdot y$ (i.e., replace "$>$" with "$\leqslant$") from Chang et al.'s original definition.[2]

Definition 1 [8] Optimally linearly ordered sets. A collection of sets $S = \{S_1, S_2, \ldots, S_m\}$, where each set consists of d-dimensional point objects, is *optimally linearly ordered* if and only if, given any d-dimensional vector p,

$$\exists x \in S_i \ s.t.$$
$$\forall y \in S_{i+j} \quad \text{and} \quad j > 0, \quad p \cdot x \leqslant p \cdot y$$

where $p \cdot x$ represents the *inner product* of the two vectors p and x.

Chang et al. [8] have proved that a set of layers constructed out of convex hulls is optimally linearly ordered. That is, in the ith layer, there is at least one tuple whose score for any preference vector p is lower than or equal to those of all the tuples in subsequent layers. Thus, top-k queries are answered by reading tuples in at most k layers. ONION is capable of answering the query with a non-monotone linear score function as well because of the geometrical properties of the convex hull. On the other hand, the query performance is adversely affected by relatively large layer sizes [22].

2.1.2. AppRI

AppRI [22] improves query performance by reducing layer sizes. AppRI constructs a database in more layers than ONION does taking advantage of the assumption that queries have monotone linear score functions. For arbitrary monotone queries, AppRI exploits the domination relation of skylines [6] to compute whether the score of the tuple a is higher or lower than that of the tuple b. Using this concept, AppRI constructs the database in much finer layers than ONION does by precomputing the domination relation for all the pairs of tuples in the database. However, due to the high time complexity of the algorithm that exactly computes the domination relation, Xin et al. [22] uses an approximate algorithm. This approximation makes layer sizes larger for more-than-three dimensional databases, resulting in poor query performance.

2.2. List-based method

The list-based method constructs a set of lists by sorting all tuples based on their values in each attribute. It then finds the top-k tuples by merging as many lists as are needed [2,11]. For example, the threshold algorithm (TA) [11], a list-based method, sequentially accesses each sorted list in parallel. That is, it accesses the first element of each sorted list, then the second

[2] In the original definition, the inequality sign should have been "$\geqslant$" instead of "$>$". Not all the boundary tuples of a convex hull are vertices [13]. Those boundary tuples that are not vertices can optimize some linear functions [13]. Thus, for a specific preference vector p, the maximum score of the tuples in the ith layer must be higher than or equal to those of all the tuples in subsequent layers.

element, and so on, until a particular threshold condition is met. For each tuple identifier seen under the sorted accesses, it also randomly accesses the other lists to get its values of the other attributes to compute the tuple's score. TA and its variants are useful for a distributed database environment where exploiting the attribute correlation is difficult due to vertical partitioning [11]. However, the performance of these methods are penalized due to lack of exploiting the attribute correlation [8,22].

2.3. View-based method

The basic idea behind the view-based method is to "precompute" the answers to a class of queries with different preference vectors and return the precomputed top-k answers given a query. When the exact answers to the query issued by a user has not been precomputed, the "closest" precomputed answers are used to compute the answer for the query. PREFER [15] and LPTA [10] are well-known methods of this approach. Because the view-based approach requires constructing an index for each query, its space and maintenance overhead often becomes an important issue in using this approach for a practical system [21].

2.4. Other methods

There exists a large body of work for efficient computation of skyline queries [9,17,19,20]. Because the skyline contains at least one tuple that minimizes any monotone scoring function [6], this body of work can be used to deal with top-k queries under a monotone scoring function. In addition, there exists a body work for extending the traditional top-k queries to handle uncertain data such as the tracking data of moving objects and sensing data of sensors [5,16]. Because uncertainty of these data is modeled as a probability distribution [5,23], this body of work is focused on finding k tuples with the highest probability to be the top-k results efficiently [16]. Since uncertain data can also be modeled as fuzzy sets [24], these top-k queries on uncertain data can be further extended to handle fuzzy sets.

3. Problem definition

In this section, we formally define the problem of top-k queries. A *target relation* R has d attributes, $A_1, A_2, \ldots, A_d$ of real values, and the cardinality of R is N [10,15,22]. Every tuple in the relation R can be considered as a point in the d-dimensional space $[0.0, 1.0]^d$. Hereafter, we call the space $[0.0, 1.0]^d$ as the *universe*, refer to a tuple t in R as an object t in the universe, and use the tuple and the object interchangeably as is appropriate. Table 1 summarizes the notation to be used in this paper.

A *top-k query* Q is defined as a pair $f(t)$ and k, where $f(t)$ is a score function such that $f(t) = \sum_{i=1}^{d} p[i] * t[i]$, and k is the number of result objects. Here, as summarized in Table 1, $p[i]$ denotes the weight, called the *attribute preference* [15], of the ith attribute, and $t[i]$ denotes the value of the ith attribute of the object $t \in R$. Since $f(t)$ is *monotone*, $p[i] \geqslant 0 (1 \leqslant i \leqslant d)$. We assume that $p[i]$'s are normalized so that $\sum_{i=1}^{d} p[i] = 1$.

As the result of top-k queries, k objects with the lowest (or highest) scores for the score function are retrieved. Without loss of generality, we assume that we are looking for the lowest-scored objects in the rest of this paper. Therefore, our goal is to retrieve a sequence of objects $[t^1, t^2, \ldots, t^k]$ that satisfy $f(t^1) \leqslant f(t^2) \leqslant \cdots \leqslant f(t^k) \leqslant f(t^l)$, $k+1 \leqslant l \leqslant N$. Here, t^j denotes the jth ranked object in the ascending order of their score, where $1 \leqslant j \leqslant N$. If the user wants to retrieve the k objects with the highest scores, we can process the query by using negative values for $p[i]$ in the score function [8,22].

Table 1
The notation.

Symbols	Definitions
R	The target relation for top-k queries
N	The cardinality of R
d	The number of attributes of R or the dimension of the universe
A_i	The ith attribute of $R (1 \leqslant i \leqslant d)$
t	An object in R (t is considered as a d-dimensional vector that has $t[i]$ as the ith element)
$t[i]$	The value of attribute A_i in the object $t (t \in R)$
p	A preference vector [15] (a d-dimensional vector that has $p[i]$ as the ith element)
$p[i]$	The weight of attribute A_i in the preference vector p
X_i	The axis corresponding to A_i in the d-dimensional space (i.e., the universe)
δ	The offset at which objects are optimally distributed when partitioning the universe or a subregion
O	The point where the coordinate value of each axis is 0 (i.e., the origin)
$vMAX$	The point where the coordinate value of each axis in the universe is 1.0
$vMAX^+$	A virtual object whose coordinate value of the ith axis is $1.0 + \epsilon (\epsilon > 0)$
No_Objects_Read	The number of objects read from database

4. The partitioned-layer index (PL index)

4.1. Overview

We now explain how to construct the PL Index to process top-k queries. The goal of the PL Index is to reduce the sizes of the layers that are read for processing top-k queries because the layer size affects query performance in layer-based methods as explained in Introduction.

To achieve the goal, the PL Index is built through the following two steps as shown in Fig. 1: (1) *Partitioning step*: partitioning the universe into a number of subregions with long narrow hyperplanes of similar sizes that are parallel to the diagonal of the universe and (2) *Layering step*: constructing the sublayer list for each subregion by using the convex skyline, which is a subset of the skyline. Here, dotted lines represent sublayers.

First, through the partitioning step, we can reduce the sublayer size because the number of objects in each subregion is smaller than that in the entire universe. The sublayer size tends to be reduced inversely proportional to the number of subregions. Besides, since our partitioning method allows us to construct sublayers whose sizes are relatively uniform in the same subregion and among different subregions, it has two desirable properties for query processing – (1) low sensitivity of query performance to the preference vector p and (2) linear dependence of query performance on k. As we will explain in Section 5, our query processing method finds query results by dynamically merging sublayer lists. That is, the sequence and the number of sublayers to be accessed are dynamically determined depending on p and k. If variation among sublayer sizes in different sublayer lists is large due to an improper partitioning method, the query performance can be sensitive to p and k. However, our query performance is not very sensitive to p and is approximately linear in the value of k by virtue of our partitioning method. Second, through the layering step, we can further reduce the sublayer size by using the convex skyline instead of using the skyline because the convex skyline is a constrained form of the skyline. Consequently, we can reduce the number of objects to be accessed for processing queries by using the PL Index. We explain in detail the partitioning step in Section 4.2 and the layering step in Section 4.3.

4.2. The partitioning step

We propose a method that partitions the universe into a number of subregions. Thus, a relation R is partitioned into a number of disjoint subsets corresponding to the subregions. Here, we partition the universe by using a hyperplane: a (straight) line when $d = 2$, a plane when $d = 3$, and a hyperplane when $d \geqslant 4$.

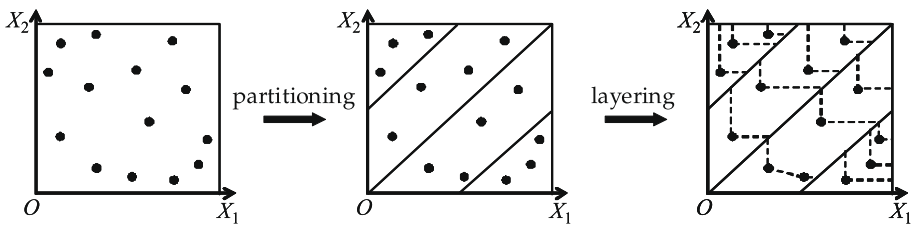


Fig. 1. Overall process for building the PL Index in the two-dimensional universe.

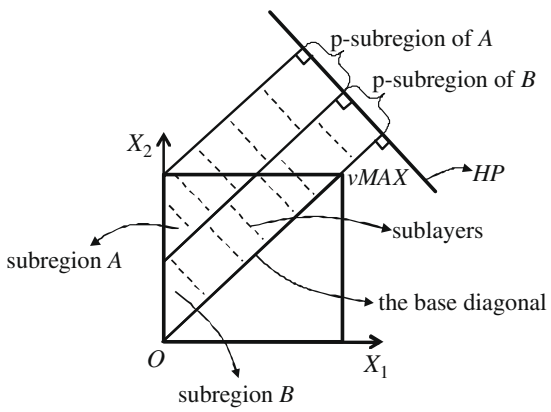


Fig. 2. Illustrating the notation in the two-dimensional universe.

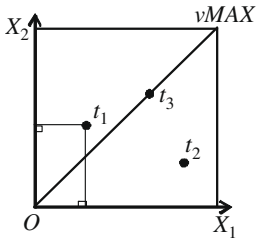


Fig. 3. The distance between an object and each axis in the two-dimensional universe.

To explain our partitioning method, we first define some terminology. We define the line segment between the origin O and *vMAX*, the point where the coordinate value of each axis in the universe is 1.0 as defined in Table 1, as the *base diagonal*. We also define a *p-subregion* as a subregion projected onto the hyperplane *HP* that is perpendicular to the base diagonal (i.e., orthographic projection). We use the concept of the p-subregion when explaining sublayer sizes. Fig. 2 illustrates the notation in the two-dimensional universe.

The query processing performance depends on the method of partitioning the universe because the sizes of sublayers, and accordingly, the numbers of objects read in from these sublayers vary depending on the sizes and shapes of the subregions. As we explained in Section 4.1, in order to make the query performance be insensitive to p and k, we need to partition the universe into subregions in such a way that the number of objects in different subregions is similar to one another. In order to do that, we partition the universe with long narrow hyperplanes of similar sizes that are parallel to the base diagonal. This method is geared to supporting monotone linear score functions. Since sublayers (i.e., convex skylines represented by dotted lines) tend to be parallel to the p-subregions as shown in Fig. 2, the sizes of the sublayers in the same subregion are relatively uniform. Besides, since the universe is partitioned into subregions where the number of objects is similar to one another, sublayer sizes in different subregions tend to be similar to one another.

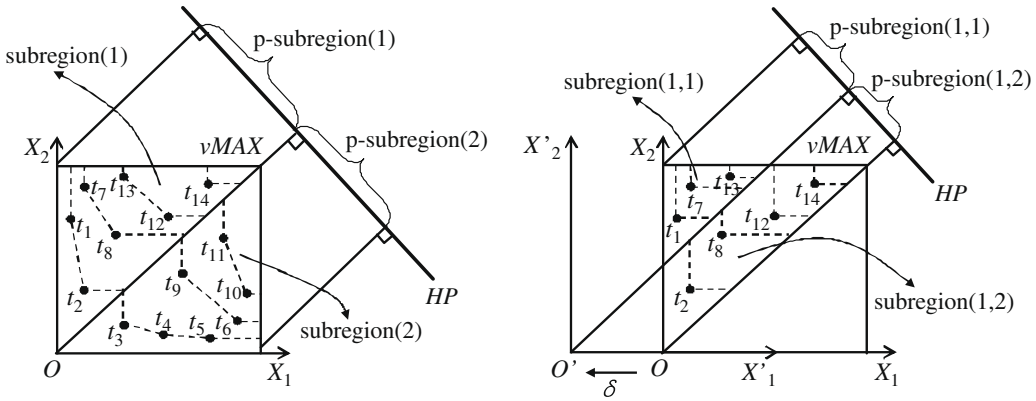
In order to partition the universe into a number of subregions (i.e., hyperplanes) that are parallel to the base diagonal, we use the distance between an object and each axis.[3] Geometrically, an object in the d-dimensional universe must be farthest from some axis among d axes (the ties are broken arbitrarily). By exploiting this property, we partition the universe into d subregions such that the objects for which X_i is the farthest axis belong to the same subregion. For instance, in Fig. 3, t_1 and t_2 are farthest from the axes X_1 and X_2, respectively. In case of t_3 that is on the base diagonal, since t_3 has the same distance to each axis, we need to break ties. In our case, we regard that t_3 is farthest from X_2 because X_2 has a larger axis number than X_1. Consequently, the two-dimensional universe containing t_1, t_2, and t_3 are partitioned into two subregions. One contains t_1, and the other contains t_2 and t_3. By translating the coordinate, we can further partition a subregion into d smaller subregion recursively. In the same manner, we can partition the universe into as many subregions as are needed. We formally explain our partitioning method as follows.

An object t in the d-dimensional universe has d coordinate values $t[1], t[2], \ldots, t[d]$. If $t[i]$ is the smallest among them (if $1 \leqslant i < j \leqslant d$ and $t[i] = t[j]$, we regard $t[i]$ as the smallest), the object t is farthest from the axis X_i among the d axes. Thus, by using this property, we partition the universe into d subregions such that the objects for which X_i is the farthest axis belong to the same subregion. Hereafter, we call the region where any object t in it satisfies the condition $t[i] = min_{l=1}^{d}(t[l])$ as *subregion*(i).

To further partition subregion (i) into d smaller subregions we do the following. We first translate the coordinates by shifting the origin O along the negative direction of the axis X_i by δ (>0.0) since the objects in subregion(i) exist only towards the negative direction of the axis X_i. (For example, see Figs. 4 and 5.) Here, δ is the offset at which the standard deviation of the numbers of the objects in d smaller subregions of subregion(i) is the smallest. We denote the new origin and axes as O' and X'_i $(1 \leqslant i \leqslant d)$, respectively. Let $t'[1], t'[2], \ldots, t'[d]$ be the coordinate values of the object t in subregion(i) in the new coordinates. We next recursively partition subregion(i) into d smaller subregions. Hereafter, we call the region where any object t' in it satisfies the condition $t'[j] = min_{l=1}^{d}(t'[l])$ as *subregion*(i,j). Consequently, if we recursively partition the universe h times in the same manner, the universe is partitioned into d^h subregions.

Example 1. Fig. 4 shows an example of partitioning the two-dimensional universe. As shown in Fig. 4a, we first partition the set of objects $t_1, \ldots, t_{14}$ in the universe into two subregions. Since t_1, t_2, t_7, t_8, t_{12}, t_{13}, and t_{14} are farthest from the axis X_1, they belong to subregion(1). Similarly, t_3, t_4, t_5, t_6, t_9, t_{10}, and t_{11} belong to subregion(2) because they are farthest from the axis X_2. Here, dotted lines represents sublayers (i.e., convex skylines), and p-subregion(1) and p-subregion(2) represent those subregions projected onto the hyperplane *HP* that is perpendicular to the base diagonal. To further partition subregion(1) into two smaller subregions, we translate the coordinates by shifting the origin O along the negative direction of the axis X_1 by δ as shown in Fig. 4b. Here, X'_1, X'_2, and O' represent the axes and the origin of the new coordinates. In the new coordinates, since t_1, t_7, and t_{13} are farthest from the axis X'_1, they belong to subregion(1,1). Similarly, t_2, t_8, t_{12}, and t_{14} belong to

[3] Here, in the d-dimensional universe, the distance between the object t and the axis X_i is defined as the distance between t and the foot of perpendicular from t to the axis X_i.



(a) Partitioning the universe once. (b) Partitioning subregion(1).

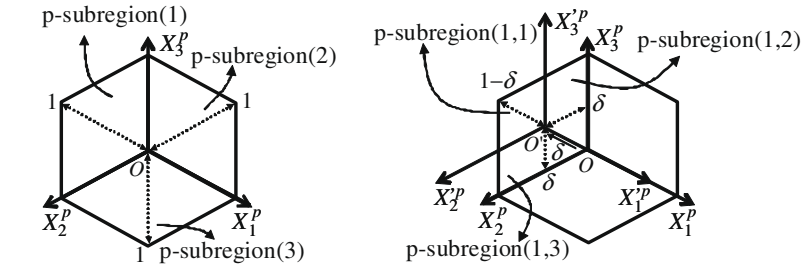
Fig. 4. An example of partitioning the two-dimensional universe.

subregion(1,2) because they are farthest from the axis X'_2. Likewise, subregion(2) is further partitioned into subregion(2,1) and subregion(2,2). We note that subregions are parallel to the base diagonal, and the sizes of sublayers in the same subregion are similar to one another since they are expected to be proportional to the sizes of p-subregions.

Example 2. Fig. 5 shows an example of p-subregions after partitioning the three-dimensional universe. Here, we represent the axes X_i $(1 \leqslant i \leqslant 3)$ in the three-dimensional universe that are projected onto the two-dimensional plane that is perpendicular to the base diagonal as X_i^p. Figs. 5a and b represent the universe projected onto the two-dimensional plane. In Fig. 5a, p-subregion(1), p-subregion(2), and p-subregion(3) represent p-subregions for the subregions that are obtained by partitioning the universe once. The dotted line with arrow head represents the width of the universe starting from the origin O to the negative limit of each axis. The widths for all axes are 1. In Fig. 5b, p-subregion(1,1), p-subregion(1,2), and p-subregion(1,3) represent p-subregions for the subregions that are obtained by further partitioning subregion(1). In order to partition subregion(1), we translate the coordinates by shifting the origin O along the negative direction of the axis X_1 by δ. Here, we denote the new origin and axes as O' and X'_i $(1 \leqslant i \leqslant 3)$. The width of subregion(1) for the axis X'_1 is $1-\delta$, and those for the other axes are δ. Similarly, subregion(2) and subregion(3) are further partitioned.

Fig. 6 shows the algorithm *BasicUniversePartitioning* for partitioning the universe. The inputs to BasicUniversePartitioning are a set S of d-dimensional objects and the *partition level* h $(\geqslant 1)$, which represents the number of times the universe has been recursively partitioned. The current partition level i and the origin o of the new coordinates for translation have been initialized with 0 and with the origin O, respectively. The output is the set $\{S_1, S_2, \ldots, S_c\}$ of $c = d^h$ disjoint subsets of S. In Step 1, data structures are initialized. In Step 2, S is partitioned into d disjoint subsets $T_1, T_2, \ldots, T_d$ according to the farthest axis for each object in S when the objects in S are translated by the new origin o. In Step 3, if the current partition level i is equal to the partition level h, it returns $\{T_1, T_2, \ldots, T_d\}$ as the results. Otherwise, it recursively calls itself for each $T_j (1 \leqslant j \leqslant d)$ in order to further partition T_j. Here, δ is subtracted from $o[j]$ in order to shift the origin along the negative direction of the axis X_j by δ. For ease of understanding, we first fix δ to be $\frac{1}{2^i}$.

We then extend this basic partitioning algorithm to use the optimal δ. Here is the sketch of the algorithm. (1) Initial partitioning: Over the entire universe, we find the new origin o where the standard deviation of the sizes of d subsets of the universe is the smallest, and then, partition the universe into d subregions with the new origin. (2) Recursive partitioning: We further recursively partition each subregion into d smaller subregions by shifting the coordinates by δ. Here, δ is the point



(a) Partitioning the universe once. (b) Partitioning subregion (1).

Fig. 5. An example of p-subregions after partitioning the three-dimensional universe.

Algorithm *BasicUniversePartitioning*:

Input: (1) S: a set of d-dimensional objects
(2) h: the partition level /* $h \geq 1$ */
(3) i: the current partition level /* initialized with 0 */
(4) o: the origin of the new coordinates /* initialized with the origin O */

Output: $\{S_1, S_2, \ldots, S_c\}$: the set of $c = d^{(h-i)}$ disjoint subsets of S

Algorithm:

Step 1. Initialization: FOR $j := 1$ to d DO $T_j := \{\}$
Step 2. Partition: FOR $t \in S$ DO /* distribute objects */
2.1 Find j such that $t[j] - o[j] = min_{i=1}^{d}(t[i] - o[i])$
/* find the farthest axis X_j for $t' = t - o$ in the new coordinates */
2.2 $T_j := T_j \cup \{t\}$
Step 3. Recursive Partitioning
3.1 $i := i + 1$ /* current partition level */
3.2 IF $i = h$ RETURN $\{T_1, T_2, \ldots, T_d\}$
3.3 ELSE
3.3.1 FOR $j := 1$ to d DO /* for each axis X_j */
3.3.1.1 $\delta := 1.0 / 2^i$ /* the offset at which objects are distributed */
3.3.1.2 $o_j[j] := o[j] - \delta$ /* the origin for the new coordinates for partitioning the subregion containing T_j */
3.3.1.3 $U_j := BasicUniversePartitioning(T_j, h, i, o_j)$

Fig. 6. The basic universe partitioning algorithm.

where the standard deviation of the numbers of the objects in d smaller subregion is the smallest. In practice, we find the optimal δ by probing δ with a fixed increment $w[i]/m$. Here, $m(> 0)$ is the number of points probed, and $w[i]$ is the width of the subregion starting from the origin to the negative limit of the axis X_i. The detailed algorithm for this extension is in Fig. 7, and its functions are in Fig. 22 of Appendix A.

4.3. The layering step

For each subregion obtained in the partitioning step, we construct the list of sublayers over objects in the subregion. We use the *convex skyline* that we formalize in Definition 3 to construct a sublayer. The convex skyline is computed from the skyline and the convex hull. The convex skyline has the characteristic of containing the object with the lowest score (i.e., the top-1 object) for an arbitrary *monotone linear* function. We prove it in Lemma 3.

Before we formally define the convex skyline, we provide some basic intuition. As we have mentioned in Section 3, we have assumed that queries have monotone linear functions and further assumed that we are finding the minimum-scored objects. (1) By the former assumption, we do not need to support non-monotone linear functions or monotone non-linear functions. Thus, we combine the convex hull and the skyline because the convex hull is a useful notion when supporting (monotone or non-monotone) *linear* functions [8]; the skyline (linear or non-linear) *monotone* functions [6] (see Definition 2 and Lemma 1 below). (2) By the latter assumption, we do not need to find those objects that only maximize an arbitrary monotone linear score function. In order to eliminate those objects, we use $vMAX^+$, a virtual object whose coordinate value of the ith axis is $1.0 + \epsilon$ ($\epsilon > 0$) as defined in Table 1, when computing the convex hull. $vMAX^+$ has this effect since the coordinate values of $vMAX^+$ are strictly larger than those of any object (including $vMAX$) in the universe. We prove it in Lemma 2.

Definition 2. [6] (Skyline) The *Skyline* is defined as those points that are not dominated by any other point. A point A *dominates* another point B if A is as good as or better than B in all dimensions and better than B in at least one dimension.

Lemma 1. [6] *The skyline constructed over a set S of objects in the universe contains at least one object that minimizes an arbitrary monotone function over the objects in S.*

Lemma 2. *Let f be an arbitrary monotone linear score function. For a set S of objects in the universe, $vMAX^+$ is the only object that maximizes f over the objects in $S \cup \{vMAX^+\}$.*

Proof. See Appendix B. □

Algorithm *UniversePartitioning*:

Input: (1) S: a set of d-dimensional objects
(2) h: the partition level /* $h \geq 1$ */
(3) o: the origin of the coordinates
(4) w: the width of the universe containing S starting from the origin to the negative limit of the coordinates
/* w is initialized to be <1.0, …, 1.0> */

Output: $\{S_1, S_2, \ldots, S_c\}$: the set of $c = d^h$ disjoint subsets of S

Algorithm:

1. Over the entire universe, find the new origin o' where the standard deviation of the sizes of d subsets of S that are obtained by calling *DistributeObjects*(S, o') is minimized
2. $w' := w + o - o'$ /* the new width of the entire universe containing S in the new coordinates */
3. $i := 0$ /* current partition level */
4. RETURN *RecursivePartitioning*(S, h, i, o', w')

Function *RecursivePartitioning*( )

Input: (1) S: a set of d-dimensional objects
(2) h: the partition level /* $h \geq 1$ */
(3) i: the current partition level
(4) o: the origin of the new coordinates
(5) w: the width of the region containing S

Output: $\{S_1, S_2, \ldots, S_c\}$: the set of $c = d^{(h-i+1)}$ disjoint subsets of S

1. $\{T_1, \ldots, T_d\} := DistributeObjects(S, o)$ /* partitioning */
2. $i := i + 1$ /* current partition level */
3. IF $i = h$ RETURN $\{T_1, T_2, \ldots, T_d\}$
4. ELSE
   4.1 FOR $j := 1$ to d DO /* for each axis X_j */
      4.1.1 $\delta_j := FindOptimalOffset(j, T_j, o, w)$
      4.1.2 $o_j[j] := o[j] - \delta_j$ /* the origin for the new coordinates for optimally partitioning the subregion containing T_j */
      4.1.3 FOR $k := 1$ to d DO /* the width of the subregion containing T_j */
         IF $k = j$ DO $w_j[j] := w[j] - \delta_j$
         ELSE $w_j[k] := \delta_j$
      4.1.4 $U_j := RecursivePartitioning(T_j, h, i, o_j, w_j)$
   4.2 RETURN $U_1 \cup U_2 \cup \ldots \cup U_d$

Fig. 7. The universe partitioning algorithm extended to use the optimal δ.

Corollary 1. *$vMAX^+$ does not minimize f (when $|S| > 0$).*

Definition 3. (Convex skyline) We define $SL(S)$ as the set of objects in the skyline constructed over a set S of objects in the universe. We then define $CH(T)$ as the set of objects in the convex hull vertices over $T = SL(S) \cup \{vMAX^+\}$ [8]. We define $CH(T) - \{vMAX^+\}$ as the *convex skyline* $CS(S)$.

Example 3. Fig. 8 shows the convex skyline constructed over the set $S = \{t_1, t_2, \ldots, t_9\}$ of the two-dimensional objects. The solid line represents the skyline $SL(S) = \{t_2, t_3, t_4\}$. The dotted line represents the convex hull vertices $CH(T) = \{t_2, t_4, vMAX^+\}$ constructed over $T = SL(S) \cup \{vMAX^+\}$. By Definition 3, the thick shaded line represents the convex skyline $CS(S) = \{t_2, t_4\}$ (i.e., $\{t_2, t_4, vMAX^+\} - \{vMAX^+\}$).

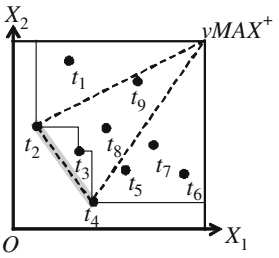


Fig. 8. An example of finding the convex skyline over a set of two-dimensional objects.

Lemma 3. *Let f be an arbitrary monotone linear score function. For a set S of objects in the universe, $\exists t \in CS(S)$ such that t minimizes f over the objects in S.*

Proof. See Appendix C. □

For an arbitrary monotone linear score function f, at least one object that minimizes f exists in a skyline [6], and at least one exists in the convex skyline as well. The convex skyline, however, is a more constrained form of the skyline. Thus, we can reduce the sublayer size by using the convex skyline instead of using the skyline according to Lemma 4. That is, $CS(S) \subseteq SL(S)$, and thus, $|CS(S)| \leqslant |SL(S)|$.

Lemma 4. *Let f be an arbitrary monotone linear score function. For a set S of objects in the universe, $\neg\exists t \in CS(S)$ such that $t \in SL(S)$ and t only maximizes f over the objects in $SL(S)$.*

Proof. See Appendix D. □

Lemma 4 does not guarantee that all the objects in $CS(S)$ minimize a monotone linear score function over the objects in S. For example, an object that minimizes or maximizes a non-monotone linear score function over the objects in $SL(S)$ can exist in $CS(S)$. We note that we eliminate only those objects that only maximize an arbitrary monotone linear score function over the objects in $SL(S)$ from $CS(S)$. In this sense, the convex skyline is not minimal, yet is effective in reducing the size of the layer.

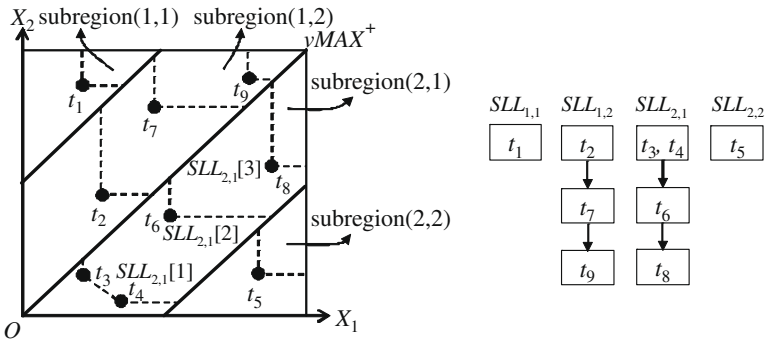
Example 4. In Fig. 8, $SL(S) = \{t_2, t_3, t_4\}$ and $CS(S) = \{t_2, t_4\}$. Thus, $CS(S) \subseteq SL(S)$.

We now discuss the method for building sublayer lists by using the convex skyline. For a given set S of objects, we construct the sublayer list by repeatedly finding convex skylines over the remaining objects. That is, we make the first sublayer $L[1]$ with the convex skyline over the set S, and then, make the second sublayer $L[2]$ with the convex skyline over $S - L[1]$, and so on. The sublayer list constructed over each subregion satisfies the optimally linearly ordered property [8]. We prove it in Lemma 5.

Lemma 5. *For a given set S of objects in the universe, a sublayer list constructed using convex skylines over S is optimally linearly ordered as defined in Definition 1.*

Algorithm *LayerlistBuilding*:
Input : (1) S: a set of d-dimensional objects
Output: L: the list of layers (convex skylines)
Algorithm:
1. WHILE $S \neq \{\}$ DO /* constructing layers */
   1.1 Find the skyline $SL(S)$ over S
   1.2 $T := SL(S) \cup \{vMAX^+\}$
   1.3 Find the convex hull_V $CH(T)$ over T
   1.4 $i := i + 1$ /* layer number, i is initialized with 0 */
   1.5 $L[i] := CH(T) - \{vMAX^+\}$ /* convex skyline */
   1.6 $S := S - L[i]$
2. RETURN $L := L[1]\, L[2] \ldots L[i]$

Fig. 9. The LayerlistBuilding algorithm for constructing a sublayer list with convex skylines.



(a) The sublayerlists for the four subregions. (b) The PL Index.

Fig. 10. An example of constructing the sublayer lists and the PL Index in the two-dimensional universe.

Proof. See Appendix E. □

Fig. 9 shows the algorithm *LayerlistBuilding* for constructing a list of sublayers over a given set S of the d-dimensional objects. The input to LayerlistBuilding is the set S of the d-dimensional objects, and the output is the list L of convex skylines. In Step 1, the algorithm iteratively finds the convex skyline $L[i]$ over S until S is empty. According to Definition 3, in Steps $1.1 \sim 1.5$, it finds the convex hull vertices over $SL(S) \cup \{vMAX^+\}$, and then, eliminates $vMAX^+$ from the convex hull vertices found. In Step 2, it returns the list of convex skylines $L[1], L[2], \ldots, L[i]$ as the result.

Example 5. Fig. 10 shows an example of constructing the sublayer lists in the two-dimensional universe. Fig. 10a shows the sublayer lists for four subregions subregion(1,1), subregion(1,2), subregion(2,1), subregion(2,2), which are obtained by partitioning the two-dimensional universe. The LayerlistBuilding algorithm constructs a sublayer list for the set of objects in each subregion. For the subregion(2,1), it constructs $SLL_{2,1}[1] = \{t_3, t_4\}$ as the first sublayer, $SLL_{2,1}[2] = \{t_6\}$ over the remaining objects as the second sublayer, and $SLL_{2,1}[3] = \{t_8\}$ as the third sublayer. Fig. 10b shows the PL Index for the subregions shown in Fig. 10a. The lists $SLL_{1,1}$, $SLL_{1,2}$, $SLL_{2,1}$, and $SLL_{2,2}$ correspond to the sublayer lists for subregion(1,1), subregion(1,2), subregion(2,1), and subregion(2,2), respectively.

5. The query processing algorithm using the PL Index

As shown in Fig. 10b, the PL Index can be considered as a set of ordered lists of sublayers. That is, as mentioned in Lemma 5, there is an order among sublayers within a sublayer list, but there is none among sublayers in different sublayer lists. Thus, to process queries using the PL Index, we merge the sublayer lists by sequentially reading sublayers from their heads. To find the top-1 object, we read the first sublayers of all the sublayer lists because we do not know which of them contains the top-1 object. To find top-i object ($2 \leqslant i \leqslant k$), we need to read the subsequent sublayer of the sublayer list containing the top-$(i-1)$ object as the minimum-scored object in the sublayer because in this case it is not guaranteed that the sublayer where the top-$(i-1)$ object has been found contains the top-i object.

Query processing involves two different procedures depending on the value of i: (1) finding the top-1 object and (2) finding the top-i object ($2 \leqslant i \leqslant k$). To find the top-1 object, we read the first sublayers of all the sublayer lists. By Lemma 5, for any monotone linear score function, the object with the lowest score in each sublayer list resides in the first sublayer. Since we do not know which of the first sublayers contains the top-1 object, we have to read the objects in all the first sublayers into memory, and then, find the top-1 object by computing the scores of these objects. To find the top-i object ($2 \leqslant i \leqslant k$), we read at most one sublayer from a specific sublayer list. This is because each sublayer list satisfies the optimally linearly ordered property. In Theorem 1 below, we prove the principle of finding the top-i object ($2 \leqslant i \leqslant k$). Here, if we know which sublayer contains the top-$(i-1)$ object, we can also know which sublayer should additionally be read to find the top-i object ($2 \leqslant i \leqslant k$). We first summarize in Table 2 the notation to be used for proving Theorem 1.

Table 2
Notation for proving Theorem 1.

Symbols	Definitions
found_list(i)	The sublayer list containing the top-i object
current_sublayer(i)	The sublayer that is the most recently read in *found_list*(i)
next_sublayer(i)	The next sublayer of *current_sublayer*(i)
min_object(l)	The object with the lowest score in the sublayer l
next_object(t, L)	The object with the lowest score next to the object t in the sublayer list L

Theorem 1. *Let us add objects in the first sublayer of each sublayer list to a set H (the added objects reside both in the sublayers and in H). If the top-$(i-1)$ object $(2 \leqslant i \leqslant k)$ is min_object(current_sublayer$(i-1)$), the top-i object exists in $H \cup$ next_sublayer$(i-1)$. Otherwise, the top-i object exists in H.*

Proof. We refer to the top-*i* object as the top-*i*. (1) In the case of $i = 2$, the top-2 is *next_object*(top-1, *found_list*(1)) or the object (which must be in *H* already) with the lowest score in a sublayer list that does not contain the top-1. According to Lemma 5, in the first sublayer of *found_list*(1), there is at least one object whose score is lower than or equal to those of all the objects in all subsequent sublayers. When there is only one such object in the first sublayer, that object is the top-1, and *next_object*(top-1, *found_list*(1)), which can possibly be the top-2, exists in *next_sublayer*(1). Consequently, we add objects in *next_sublayer*(1) to *H*. (2) In case of $i \geqslant 3$, the top-*i* is either *next_object*(top-$(i-1)$, *found_list*$(i-1)$) or the object (which must be in *H* already) with the lowest score next to the top-$(i-1)$ in a sublayer list that does not contain the top-$(i-1)$. In this case, *next_object*(top-$(i-1)$, *found_list*$(i-1)$) can be found in one of the two places:

Case 1: top-$(i-1)$ = *min_object*(*current_sublayer*$(i-1)$): As in the case of $i = 2$, according to Lemma 5, *next_object*(top-$(i-1)$, *found_list*$(i-1)$) may reside in *next_sublayer*$(i-1)$. Thus, the top-*i* exists in $H \cup$ *next_sublayer*$(i-1)$.

Case 2. top-$(i-1) \neq$ *min_object*(*current_sublayer*$(i-1)$): According to Lemma 5, the score of *min_object* (*current_sublayer*$(i-1)$) is lower than or equal to those of all the other objects in subsequent sublayers of *current_sublayer*$(i-1)$. Because *min_object*(*current_sublayer*$(i-1)$) has not been returned as a result yet, we do not need to add objects in *next_sublayer*$(i-1)$, which must have scores higher than or equal to that of *min_object*(*current_sublayer*$(i-1)$), to *H*. Thus, the top-*i* exists in *H* $(2 \leqslant i \leqslant k)$.

Consequently, only when the top-$(i-1)$ is the same as *min_object*(*current_sublayer*$(i-1)$), we need to add objects in *next_sublayer*$(i-1)$ to *H*. Otherwise, we can find the top-*i* in *H*. □

As we discussed above, when processing queries using the PL Index, the sequence of sublayers to be accessed for a given query is dynamically determined. This characteristic distinguishes our method from existing layer-ordering methods [8,22] in which the sequence of layers to be accessed is fixed regardless of queries. Hereafter, we call the query processing algorithm using the PL Index *DynamicLayerOrdering*.

We now explain additional data structures we keep in memory used by the DynamicLayerOrdering algorithm. Fig. 11 illustrates the heap *H* and the array *A*. We consider that the PL Index consists of *c* sublayer lists. The heap *H* is used for finding *k* objects with the lowest scores among the objects read from each sublayer list. The heap *H* needs to maintain at least *k* objects with the lowest scores. The root of the heap *H* contains the object with the lowest score among the objects in *H* (i.e., a *min heap* [14]). Hence, by repeating the procedure of deleting the object in the root of *H* and readjusting *H*, we can incrementally find the object from the root of *H* in a sorted order. Hereafter, for simplicity of the explanation, we assume that the heap *H* is automatically readjusted when an object is deleted (added) from (to) *H*. Array *A* is used for storing the object with the lowest score in the sublayer that is most recently read (i.e., *current_sublayer*) from each sublayer list. We use array *A* to check whether or not top-$(i-1)$ is the same as *min_object*(*current_sublayer*$(i-1)$) (Theorem 1).

Fig. 12 shows the DynamicLayerOrdering algorithm. The inputs to DynamicLayerOrdering are the PL Index that consists of *c* sublayer lists and a query $Q = (f(), k)$. The output is the sequence of *k* objects having the lowest scores for the score function $f()$. In Step 1, the algorithm finds top-1. It first adds to the heap *H* the objects in $L_j[1] (1 \leqslant j \leqslant c)$ – the first sublayer of each sublayer list. Here, while adding the objects in $L_j[1]$ to *H*, the algorithm assigns the object with the lowest score in $L_j[1]$ for $f()$ (i.e., *min_object*$(L_j[1])$) to $A[j]$. Next, it returns the object in the root of *H* as top-1 and deletes the object from *H*. In Step 2, the algorithm finds top-*i* $(2 \leqslant i \leqslant k)$. In this step, we need to check whether or not top-$(i-1)$ is the same as *min_object*(*current_sublayer*$(i-1)$). Let us assume that the sublayer list containing the top-$(i-1)$ is L_m $(1 \leqslant m \leqslant c)$. If top-$(i-1) = A[m]$ (i.e., *min_object*(*current_sublayer*$(i-1)$)), the algorithm adds the objects in *next_sublayer*$(i-1)$ to *H*. Here, as in Step 1, the algo-

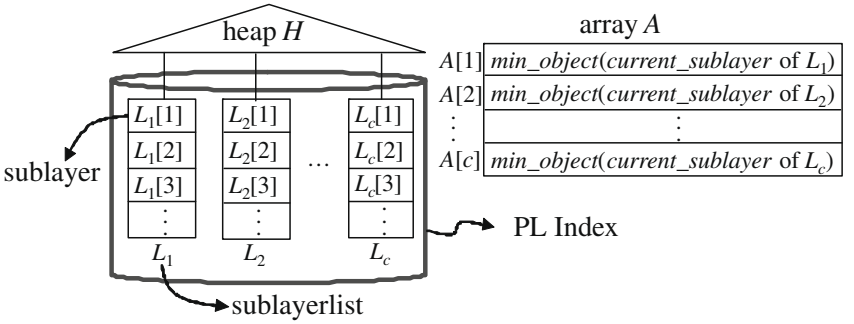


Fig. 11. The data structures in memory used by the DynamicLayerOrdering algorithm.

Algorithm *DynamicLayerOrdering*:

Input: (1) $\{L_1, L_2, \ldots, L_c\}$: a PL Index consisting of c sublayer lists
(2) $Q = (f(), k)$: a top-k query with a monotone score function $f()$

Output: [top-1, top-2, …, top-k]: the sequence of k objects with the lowest scores for $f()$

Algorithm:

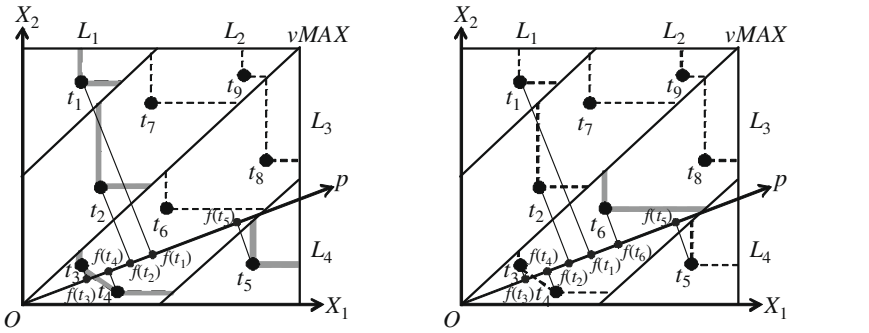
Step 1. Finding top-1:
1.1 FOR $j := 1$ to c DO
1.1.1 Add the objects in the sublayer $L_j[1]$ to the heap H
1.1.2 $A[j] := min\_object(L_j[1])$
1.2 RETURN the object in the root of H as top-1; delete the object from H

Step 2. Finding top-i: FOR $i := 2$ to k DO
2.1 $m :=$ the sequence number of the sublayer list containing top-(i-1)
/* $1 \le m \le c$ */
2.2 IF top-(i-1) = $A[m]$ DO (Theorem 1)
2.2.1 Add the objects in $next\_sublayer(i\text{-}1)$ to H
2.2.2 $A[m] := min\_object(next\_sublayer(i\text{-}1))$
2.3 RETURN the object in the root of H as top-i; delete the object from H

Fig. 12. The DynamicLayerOrdering algorithm for processing top-k queries using the PL Index.

rithm assigns the object with the lowest score in $next\_sublayer(i-1)$ for $f()$ (i.e., $min\_object(next\_sublayer(i-1))$) to $A[m]$. Finally, it returns the object in the root of H as top-i and deletes the object from H.

Example 6. Fig. 13 shows an example of processing a top-k query over a set of point objects $R = \{t_1, t_2, \ldots, t_9\}$ in the two-dimensional universe. Fig. 13a shows the sublayer lists $L_1, \ldots, L_4$ for four subregions. That is, $L_1 = [\{t_1\}]$, $L_2 = [\{t_2\}, \{t_7\}, \{t_9\}]$, $L_3 = [\{t_3, t_4\}, \{t_6\}, \{t_8\}]$, and $L_4 = [\{t_5\}]$. Here, $k = 3$, and the preference vector $p = \langle 0.7, 0.3 \rangle$. First, in order to find top-1, the objects in the first sublayer of each sublayer list are read into H. In Fig. 13a, the objects t_1 of $L_1[1]$, t_2 of $L_2[1]$, t_3 and t_4 of $L_3[1]$, and t_5 of $L_4[1]$ on the shaded lines represent those objects that are read into H. $f(t_1), f(t_2), f(t_3), f(t_4)$, and $f(t_5)$ are the scores of the corresponding objects. The score of an object t is the inner product of the object t and the preference vector p, corresponding to the distance from the origin to the foot perpendicular from t to p [8]. Since $f(t_3)$ is the lowest in this case, the object t_3 of $L_3[1]$ is returned as top-1 (Lemma 5). Next, to find top-2, the objects in $L_3[2]$ (i.e, $next\_sublayer(1)$) are read into H because top-1 (i.e., t_3) = $min\_object(current\_sublayer(1))$ (i.e., t_3 of $L_3[1]$) (Case 1 in Theorem 1). In Fig. 13b, the object t_6 of $L_3[2]$ on the shaded line represents the object that is read. Since $f(t_4)$ is the lowest among the scores of the remaining objects (i.e, t_1, t_2, t_4, t_5, and t_6) in H, the object t_4 of $L_3[1]$ is returned as top-2. Finally, to find top-3, no sublayer is additionally read because top-2 (i.e., t_4 of $L_3[1]$) $\neq$ $min\_object(current\_sublayer(2))$ (i.e., t_6 of $L_3[2]$) (Case 2 in Theorem 1). Since $f(t_2)$ is the lowest among the scores of the remaining objects (i.e., t_1, t_2, t_5, and t_6), the object t_2 of $L_2[1]$ is returned as top-3.



(a) The objects on the shaded line represent those that are read to find top-1.

(b) The objects on the shaded line represent those that are read to find top-2.

Fig. 13. An example of processing a top-k query using the PL Index.

6. Analysis of the PL Index

In this section, we analyze the index building time and the query processing time of the PL Index. For ease of analysis, we use uniform object distributions.

6.1. Analysis of index building time

The time for building the PL Index is affected by the number of objects N, the dimension of the universe d, and the number of subregions c. In this section, we first present a ballpark analysis of the index building time to learn its trends of dependency on these parameters. We then compare the index building time of the PL Index with those of existing layer-ordering methods – ONION [8] and AppRI [22].

As stated in Section 4, the algorithm for building the PL Index consists of the partitioning step and the layering step. Thus, the time complexity for building the PL Index consists of those of these two steps. The time complexity of the partitioning step is $\mathcal{O}(h * m * N) = \mathcal{O}(h * N)$, where h is the partition level, because the entire database is accessed h times. Here, m is the number of probing points to obtain the optimal δ and is a constant.

The time complexity of the layering step is computed in Eqs. 2, 4 and 6. When the cardinality of the input set is n, the time complexity of finding one skyline is $\mathcal{O}(n * \ln n)$ when $d = 2$ or 3 and $\mathcal{O}(n * (\ln n)^{d-2})$ when $d \geqslant 4$ [6]. The time complexity of finding one convex hull is $\mathcal{O}(n * \ln v)$ when $d = 2$ or 3 and $\mathcal{O}\left(\frac{n}{\lfloor \frac{d}{2} \rfloor !} * v^{\lfloor \frac{d}{2} \rfloor - 1}\right)$ when $d \geqslant 4$ [1], where v is the number of objects that form the convex hull vertices. In Fig. 9, the LayerlistBuilding algorithm first finds a set of objects that form the skyline over the objects in one subregion, and then, finds a set of objects that form the convex hull vertices over a set of objects in the skyline just constructed and $vMAX^+$. Hereafter, for ease of the analysis, we assume that $vMAX^+$ is included in the skyline. First, the average number of objects in one subregion, $size_{subregion}$, is shown in Eq. (2).

$$size_{subregion} = \frac{N}{c} \tag{2}$$

Next, when the cardinality of the input set is n, the average number of objects composing one skyline is $\frac{(\ln n)^{d-1}}{(d-1)!}$ [7]. Hence, the average number of objects composing one skyline in one subregion $size_{skyline}$ is shown in Eq. (3). In the worst case, all the objects in the skyline belong to the convex hull vertices [3]. Hence, we use $size_{skyline}$ as the number of objects in one convex skyline $size_{sublayer}$ as shown in Eq. (4).

$$size_{skyline} = \frac{(\ln size_{subregion})^{d-1}}{(d-1)!} = \frac{(\ln \frac{N}{c})^{d-1}}{(d-1)!} \tag{3}$$

$$size_{sublayer} \approx size_{skyline} \tag{4}$$

Eq. (5) shows the time complexity of finding a convex skyline $time_{convex\_skyline}$. In Eq. (5), $time_{skyline}$ is the time complexity of finding a skyline over the objects in one subregion, and $time_{convex\_hull}$ is the time complexity for finding a convex hull vertices over the objects in one skyline. We note that the number of objects used in computing the skyline is greater than that in computing the convex hull vertices. Below, we show that the time complexity of computing the skyline is higher than that of computing the convex hull vertices. When $d = 2$ or 3, $time_{skyline}$ $\mathcal{O}(N * \ln N)$ is higher than $time_{convex\_hull}$ $\mathcal{O}(\ln N * \ln \ln N)$. When $d \geqslant 4$, $time_{skyline}$ $\mathcal{O}(N * (\ln N)^{c1})$ is higher than $time_{convex\_hull}$ $\mathcal{O}((\ln N)^{c2} * ((\ln N)^{c2})^{c3})$, where $c1$, $c2$, and $c3$ are positive constants. Hence, we note that $time_{convex\_skyline}$ converges to $time_{skyline}$ as in Eq. (5). Therefore, the time complexity of the layering step $time_{layering}$ is shown as in Eq. (6) because the average number of sublayers found from one subregion is $\frac{size_{subregion}}{size_{sublayer}}$.

Furthermore, since the time complexity of the layering step shown in Eq. (6) is higher than that of the partitioning step $\mathcal{O}(h * N)$ ($c = d^h$), the time complexity of the PL Index converges to that of the layering step, i.e., $time_{layering}$.

$$\begin{aligned}
time_{convex_skyline} &= time_{skyline} + time_{convex_hull} \\
&= \begin{cases} \mathcal{O}(size_{subregion} * \ln size_{subregion}) \\ +\mathcal{O}(size_{skyline} * \ln size_{sublayer}), & \text{if } d = 2,3 \\ \mathcal{O}(size_{subregion} * (\ln size_{subregion})^{d-2}) \\ +\mathcal{O}\left(\frac{size_{skyline}}{\lfloor \frac{d}{2} \rfloor !} * size_{sublayer}^{\lfloor \frac{d}{2} \rfloor - 1}\right), & \text{if } d \geqslant 4 \end{cases} \\
&= \begin{cases} \mathcal{O}(size_{subregion} * \ln size_{subregion}), & \text{if } d = 2,3 \\ \mathcal{O}(size_{subregion} * (\ln size_{subregion})^{d-2}), & \text{if } d \geqslant 4 \end{cases}
\end{aligned} \tag{5}$$

$$\begin{aligned}
time_{layering} &= c * \frac{size_{subregion}}{size_{sublayer}} * time_{convex_skyline} \\
&= \begin{cases} c * \frac{\frac{N}{c}}{\frac{(\ln \frac{N}{c})^{2-1}}{(2-1)!}} * \mathcal{O}\left(\frac{N}{c} * \ln \frac{N}{c}\right), & \text{if } d = 2,3 \\ c * \frac{\frac{N}{c}}{\frac{(\ln \frac{N}{c})^{d-1}}{(d-1)!}} * \mathcal{O}\left(\frac{N}{c} * (\ln \frac{N}{c})^{d-2}\right), & \text{if } d \geqslant 4 \end{cases} \\
&= \begin{cases} \mathcal{O}\left(\frac{N^2}{c}\right), & \text{if } d = 2,3 \\ \mathcal{O}\left(\frac{N^2 * (d-1)!}{c * \ln \frac{N}{c}}\right), & \text{if } d \geqslant 4 \end{cases}
\end{aligned} \tag{6}$$

Table 3
The time complexities of the index building algorithms.

Algorithms	Time complexities
PL Index	$\mathcal{O}\left(\frac{N^2}{c}\right), (d = 2, 3)$
	$\mathcal{O}\left(\frac{N^2*(d-1)!}{c*\ln^{d-1}}\right), (d \geqslant 4)$
ONION	$\mathcal{O}\left(\frac{N^2*(d-1)*\ln\ln N}{(\ln N)^{(d-1)}}\right), (d = 2, 3)$
	$\mathcal{O}\left(\frac{N^2}{\lfloor\frac{d}{2}\rfloor!} * (\ln N)^{(d-1)*(\lfloor\frac{d}{2}\rfloor-2)}\right), (d \geqslant 4)$
AppRI	$\mathcal{O}(2^d * N^2 * (\lceil\frac{d}{2}\rceil + \lfloor\frac{d}{2}\rfloor * \lceil\frac{d}{2}\rceil)), (d \geqslant 2)$ (AppRI(naive)) [22]
	$\mathcal{O}(2^d * N * (\ln N)^{\lceil\frac{d}{2}\rceil+\lfloor\frac{d}{2}\rfloor*\lceil\frac{d}{2}\rceil}), (d \geqslant 2)$ (AppRI(improved)) [22]

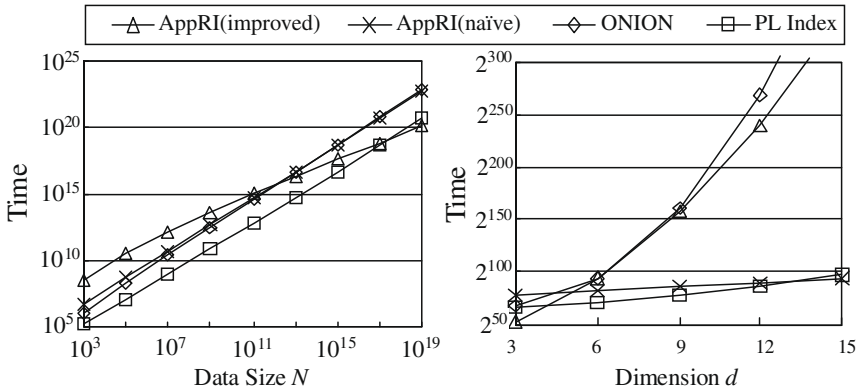
Table 3 shows the time complexities of the index building algorithms of the PL Index, ONION, and AppRI. Since Chang et al. [8] do not present the time complexity of ONION, we compute it in this paper as the product of the time complexity of finding one convex hull and the number of convex hulls constructed from N objects $\frac{N}{(\ln N)^{d-1}}$. Here, $(\ln N)^{d-1}$ is the average number of objects in one convex hull vertices [3]. Xin et al. [22] have proposed the time complexity of AppRI for the naive and improved algorithms. We call these algorithms *AppRI*(*naive*) and *AppRI*(*improved*), respectively.

Fig. 14 shows the relative execution times of those methods simulated as N or d are varied (here, h=2). Here, in order to show only the trends of complexity, we assume that the coefficients of complexity are all 1.0. Fig. 14a shows that the PL Index becomes more efficient than ONION, but worse than AppRI(improved) as N gets larger. This is because the time complexity of the PL Index, $\mathcal{O}\left(\frac{N^2}{\ln N}\right)$, is lower than that of ONION, $\mathcal{O}(N^2 * (\ln N)^{c1})$, but higher than that of AppRI(improved), $\mathcal{O}(N * (\ln N)^{c2})$, when $d \geqslant 4$ (here, $c1$ and $c2$ are a positive constant). Fig. 14b shows that the PL Index becomes more efficient than ONION, but worse than AppRI(naive) as d gets larger. Overall, the PL Index can be more efficient than ONION, whereas the PL Index can be less efficient than AppRI(improved) or AppRI(naive) depending on the dimension d.

6.2. Analysis of query processing time

Query performance of the PL Index is affected by the parameters N, d, c, and the number of query results k. In general, N, d, k are the values determined by applications. On the other hand, c is the value that can be freely tuned when creating the PL Index. In this section, we conduct a ballpark analysis of the number of subregions (i.e., c) optimizing the query performance to show the trend of dependency on k. We assume that the query processing time is proportional to the number of objects read from the database as is done in ONION [8], PREFER [15], and AppRI [22].

The query processing time $time_{query\_processing}$ when using the PL Index is obtained by using the formula shown in Eq. (7). In Step 1 of the DynamicLayerOrdering algorithm, the number of objects read is $c * size_{sublayer}$ because it reads all the objects in the first sublayers of c sublayer lists. In Step 2, while iterating loops by $(k-1)$ times, it additionally reads objects in *next_sublayer*$(i-1)$ whenever the condition top-$(i-1)$ = *min_object*(*current_sublayer*$(i-1)$) is satisfied $(2 \leqslant i \leqslant k)$. In the worst case, it always reads one sublayer for each loop. Thus, from Eqs. (3) and (4), the maximum number of objects read in Step 2 is $(k-1) * size_{sublayer}$.



(a) Time complexity as N is varied (d=5, h=2, c=d^h).

(b) Time complexity as d is varied (N=10^{11}, h=2, c=d^h).

Fig. 14. Times (simulation) of index building algorithms.

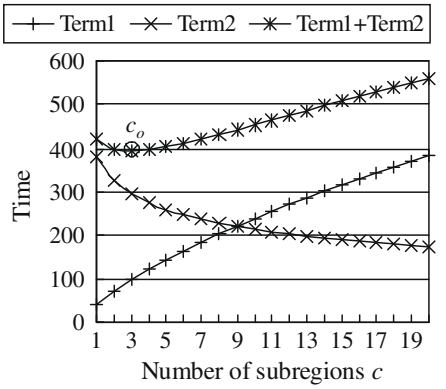


Fig. 15. Execution time (simulation) of DynamicLayerOrdering algorithm as c is varied ($N = 10$ K, $d = 3$, and $k = 10$).

$$time_{query\_processing} = c * size_{sublayer} + (k-1) * size_{sublayer} = c * \frac{(\ln \frac{N}{c})^{d-1}}{(d-1)!} + (k-1) * \frac{(\ln \frac{N}{c})^{d-1}}{(d-1)!} \quad (7)$$

In Eq. (7), we note that the query processing time increases linearly in k since the algorithm reads sublayers in proportion to k. As the number of subregions c increases, the first term of Eq. (7) increases, but the second term of Eq. (7) decreases. Specifically, since the increasing slope of the former is much steeper than the decreasing slope of the latter as shown in Fig. 15, a value of c that optimizes the query performance exists in the range of 1 to $(k-1)$. Here, when $c = k-1$, the first term is equal to the second term (i.e., $c = k-1$ is the cross-over point shown in Fig. 15). Hereafter, we refer to such a c as c_o. c_o has a tendency to increase as k does since reducing the second term of Eq. (7) is desirable. On the other hand, c_o has a tendency to decrease as k does since reducing the first term of Eq. (7) is desirable. Practically, it is not feasible to find c_o analytically for a given data set because sublayer sizes are very different depending on the distributions of data. Thus, in this paper, we experimentally find c_o's used in our experiments as shown in Section 7.2.2.

Fig. 15 shows the execution time of DynamicLayerOrdering algorithm simulated as c is varied (here, $k = 10$, $d = 3$, and $N = 10$ K). Here, in order to show only the trends of the execution time, we assume that the coefficient of the terms of Eq. (7) is 1.0. In Fig. 15, Term 1 represents the first term of Eq. (7), Term 2 represents the second term of Eq. (7), and Term 1+Term 2 represents Eq. (7). Since the increasing slope of Term 1 is 18.1 on the average over the range of $c = 1$ to 20, and the decreasing slope of Term 2 is 7.2 the average over the range of $c = 1$ to 20, we note that the former is much steeper than the latter as explained above. As shown in Fig. 15, we observe that the execution time is minimized when $c = 3$. That is, in this case, c_o exists in the range of 1 to 9.

7. Performance evaluation

7.1. Experimental data and environment

We compare the index building time and the query performance of the PL Index with those of existing layer-based methods ONION [8] and AppRI [22], and existing view-based methods PREFER [15] and LPTA [10]. For the PL Index, we use the UniversePartitioning algorithm in Fig. 7. All these methods and the PL Index except for ONION support only monotone linear score functions. Thus, for a fair comparison with ONION, we also include *Convex Skyline Index* (CSI_ONION) in our comparison as a variant of ONION specialized for monotone linear score functions. CSI_ONION constructs a single list of layers using the convex skyline instead of the convex hull vertices used in ONION. We use the wall clock time as the measure of the index building time and the number of objects read from database, *No_Objects_Read*, as the measure of the query performance. *No_Objects_Read* is a measure widely used in top-k research [8,15,22] because it is not affected by the implementation details of the individual methods. In addition, this measure is useful in environments like main memory DBMSs or the ones using flash memory (e.g., a solid-state drive (SSD)) because elapsed time is approximately proportional to the number of objects accessed in these environments where sequential/random IO cost difference is not as significant as in disk.

We perform experiments using synthetic and real data sets. For the synthetic data set, we generate UNIFORM data sets by using the generator used in AppRI [22]. The data sets consist of two-, three-, four-, five-, six-, and seven-dimensional data sets of 1 K, 10 K, and 100 K objects. For the real data set, we use a part (10,000 objects, 344 KBytes) of the Cover Forest Data[4](called *Cover3D*) (581,012 objects, 75.2MBytes) as was used by AppRI [22]. The data set consists of 10 K objects with the following three attributes: Elevation, Horizontal_Distance_To_Roadways, and Horizontal_Distance_To_Fire_Points. The correlation of these attributes in Cover3D is around 0.5, and that of the UNIFORM data sets is zero.

[4] http://www.ics.uci.edu/~mlearn/MLRepository.html.

Table 4
Experiments and parameters used for the comparison of the index building time.

Experiments		Parameters	
Exp.1	Comparison of the index building time as N is varied	Data Set	UNIFORM data
		N	1 K, 10 K, 100 K
		d	5
		h	2
Exp.2	Comparison of the index building time as d is varied	Data set	UNIFORM data
		N	10 K
		d	2, 3, 4, 5, 6, 7
		h	2
Exp.3	Comparison of the index building time as h is varied	Data set	UNIFORM data
		N	10 K
		d	5
		h	1,2,3,4

First, we compare the index building time. We measure the index building time of the PL Index, CSI_ONION, ONION, and AppRI while varying the number of objects N, the dimension d, and the partition level h. For AppRI(naive), we measure the index building time through implementation. For AppRI(improved), however, we calculate the index building time by using a formula since the implementation was not fully described by Xin et al. [22]. The formula used is (the index building time of AppRI(naive) measured $* \frac{\text{time complexity of AppRI(improved)}}{\text{time complexity of AppRI(naive)}}$). Here, we assume that the coefficients of the time complexity is 1.0 as we

Table 5
Experiments and parameters used for finding the optimal h_o.

Experiments		Parameters	
Exp.4	Finding the optimal h_o as k is varied	Data set	UNIFORM data
		N	10 K
		d	3
		k	10, 30, 50, 70, 90
Exp.5	Finding the optimal h_o as N is varied	Data set	UNIFORM data
		N	1 K, 10 K, 100 K
		d	3
		k	50
Exp.6	Finding the optimal h_o as d is varied	Data set	UNIFORM data
		N	10K
		d	2, 3, 4, 5, 6, 7
		k	50

Table 6
Experiments and parameters used for comparing the query performance.

Experiments		Parameters	
Exp.7	Comparison of the query performance as k is varied	Data set	UNIFORM data
		N	10 K
		d	3
		k	1, 10, 20, . . ., 100
Exp.8	Comparison of the query performance as N is varied	Data set	UNIFORM data
		N	1 K, 10 K, 100 K
		d	3
		k	50
Exp.9	Comparison of the query performance as d is varied	Data set	UNIFORM data
		N	10 K
		d	2, 3, 4, 5, 6, 7
		k	50
Exp.10	Comparison of the sensitivity of the query performance to the preference vector	Data set	UNIFORM data
		N	10 K
		d	3
		k	50
		–	–
Exp.11	Comparison of the query performance as k is varied using a real data set	Data set	Cover3D (real nonuniform data)
		N	10 K
		d	3
		k	1, 10, 20, . . ., 100

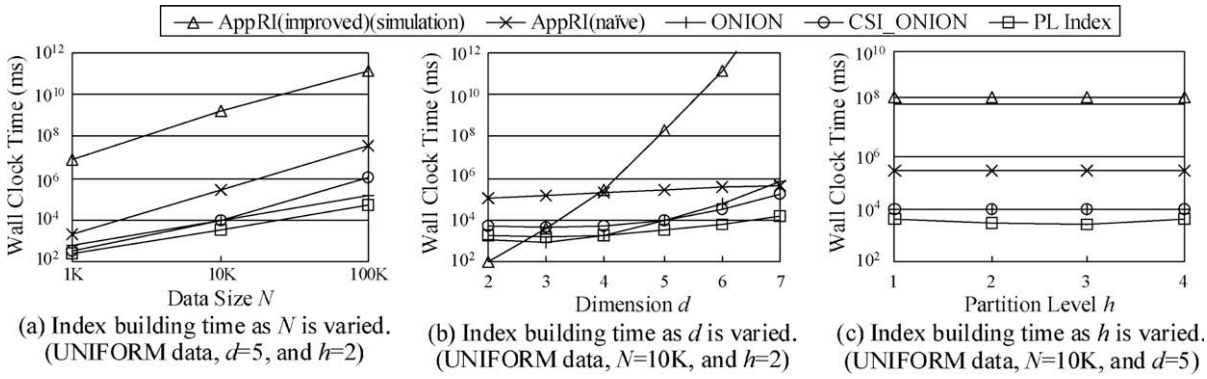


Fig. 16. The comparison of the index building time.

did in Section 6.1. We exclude PREFER and LPTA in the comparison of the index building time because these materialized-view methods do not build a separate index. Table 4 summarizes the experiments and the parameters used.

Second, before comparing the query performance, we first experimentally find the partition level h_o that optimizes the query performance of the PL Index. Hence, the total number of subregions to be used is d^{h_o}. As analyzed in Section 6.2, the query performance is affected by k, N, and d. Thus, we find the optimal partition level h_o as we vary k, N, and d. We measure *No_Objects_Read* for 100 randomly generated queries having different preference vectors, and then, use the average value over all the queries. As done in AppRI [22], the attribute preference $p[i]$, which is the weight of the ith attribute ($1 \leqslant i \leqslant d$) in the preference vector p, is randomly chosen from $\{1, 2, 3, 4\}$, and $p[i]$'s are normalized so that $\sum_{i=1}^{d} p[i] = 1$. Table 5 summarizes the experiments and the parameters used for finding the optimal partition level h_o. In Experiments 4 and 5, we use $d = 3$, which is the dimension of the main data sets that are used in AppRI [22].

Third, we compare the query performance together with its sensitivity to the preference vector. We compare the PL Index with CSI_ONION, ONION, AppRI, PREFER, and LPTA. Here, we use the optimal partition level h_o of the PL Index obtained through experiments. We measure the query performance of these methods on the synthetic and the real data sets while varying k, N, and d. We measure *No_Objects_Read* for 100 randomly generated queries, and then, average the results over all the queries. The query performance of PREFER and LPTA improves as the number of views increases. Thus, for a fair comparison with the PL Index, CSI_ONION, ONION, and AppRI that use only one index, we use only one view generated randomly. This is the same approach taken in AppRI [22]. To investigate the sensitivity, we measure *No_Objects_Read* for 100 randomly generated queries, and then, use the minimum value (MIN), the maximum value (MAX), and the average value (AVG) over all the queries. Table 6 summarizes the experiments and the parameters used. We use $d = 3$ in Experiments 7, 8, 10, and 11 as in Experiments 4 and 5.

For the experiments, we have implemented the PL Index, CSI_ONION, ONION, AppRI, PREFER, and LPTA using C++. For PREFER, we translate the code of the PREFER system[5] written in JAVA into C++. For LPTA, we used the GLPK library[6] for the implementation of linear programming. To compute convex hulls for the PL Index, CSI_ONION, and ONION, we used the Qhull library [1]. We conducted all the experiments on a Pentium-4 2.0 GHz Linux PC with 1 GB of main memory and 120 GB Segate E-IDE disks.

7.2. Results of the experiments

7.2.1. Index building time

Experiment 1: index building time as N is varied

Fig. 16a shows the index building time of the PL Index, CSI_ONION, ONION, and AppRI as N is varied from 1 K to 100 K. For the PL Index, we probe ten points for each subregion to be partitioned (i.e., m = 10 in Fig. 22) in order to find the optimal δ as explained in Section 4.3. Hereafter, we use m = 10 unless we need to change the value. As analyzed in Section 6.1, the index building time of AppRI(improved) is expected to be less than that of the PL Index when N is very large. The PL Index improves by 1.2–18.2 times over CSI_ONION, by 1.9–2.6 times over ONION, and by 7.0–582.9 times over AppRI(naive).

Experiment 2: index building time as d is varied

Fig. 16b shows the index building time as d is varied from 2 to 7. As analyzed in Section 6.1, the index building time of the PL Index becomes smaller compared with ONION and AppRI(improved) as d gets larger. The PL Index improves by 2.3–10.2 times over CSI_ONION, by 0.5–37.8 times over ONION, and by 25.3–95.9 times over AppRI(naive).

[5] http://db.ucsd.edu/PREFER/application.htm.
[6] http://www.gnu.org/software/glpk/.

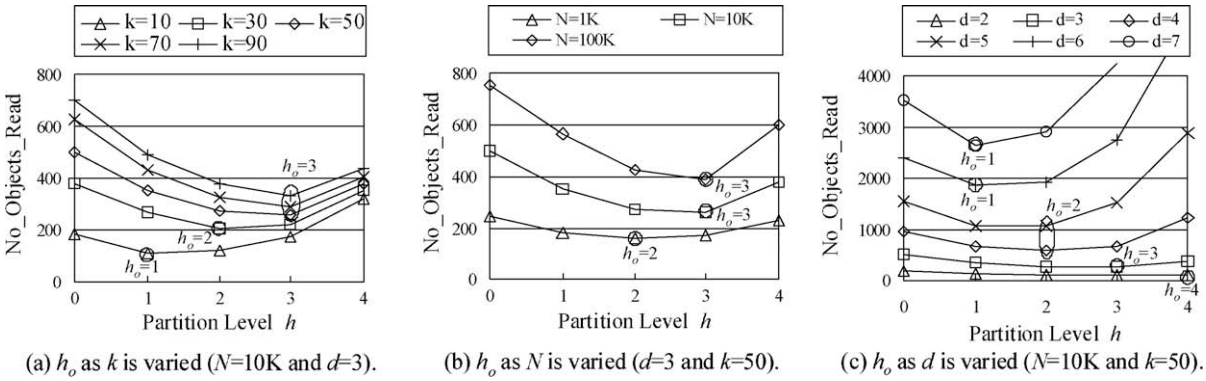


Fig. 17. The optimal partition level h_o of the PL Index (UNIFORM data).

Experiment 3: index building time as h is varied

Fig. 16c shows the index building time as h is varied from 1 to 4. We observe that for all ranges of h, the index building time of the PL Index is smaller than those of other methods. The PL Index improves by 2.0–3.2 times over CSI_ONION, by 1.9–3.1 times over ONION, and by 58.8–94.1 times over AppRI(naive).

7.2.2. Optimal partition level h_o of the PL Index

Experiment 4: h_o as k is varied

Fig. 17a shows the optimal partition level h_o that optimizes the query performance of the PL Index as k is varied. Here, $h = 0$ represents CSI_ONION, i.e., the single list of layers of convex skylines. As analyzed in Section 6.2, we note that h_o has a tendency to increase as k does since $c_o(= d^{h_o})$ increases. $h_o = 1$ when $k = 10$, $h_o = 2$ when $k = 30$, and $h_o = 3$ when k = 50, 70, and 90.

Experiment 5: h_o as N is varied

Fig. 17b shows h_o as N is varied. We observe that h_o has a tendency to increase as N does. $h_o = 2$ when $N = 1$ K and $h_o = 3$ when $N = 10$ K and 100 K.

Experiment 6: h_o as d is varied

Fig. 17c shows h_o as d is varied. We observe that h_o has a tendency to decrease as d increases. $h_o = 4$ when $d = 2$, $h_o = 3$ when $d = 3$, $h_o = 2$ when d = 4 and 5, and $h_o = 1$ when d = 6 and 7.

7.2.3. Query performance

Experiment 7: query performance as k is varied

Fig. 18a shows the query performance of the PL Index, CSI_ONION, ONION, AppRI, PREFER, and LPTA as k is varied from 1 to 100. Here, we use $h_o = 3$, which is the optimal partition level when $k = 50$. As mentioned in Section 5, when finding top-1, the query performance of the PL Index is worse than that of CSI_ONION,

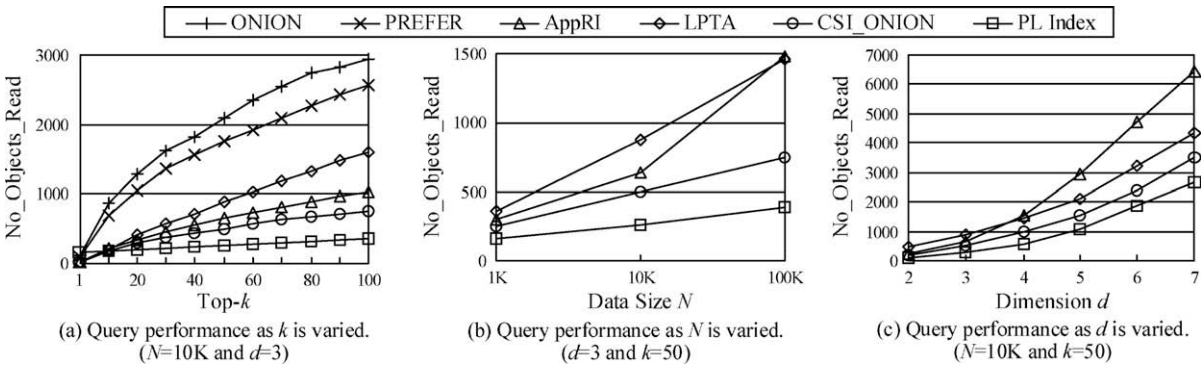


Fig. 18. The comparison of the query performance as k, N, and d are varied (UNIFORM data).

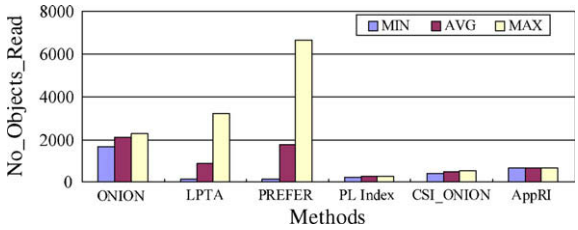


Fig. 19. The comparison of the sensitivity of the query performance to the preference vector (UNIFORM data, $N = 10$ K, $d = 3$, and $k = 50$).

ONION, AppRI, PREFER, and LPTA because the PL Index reads the first sublayer of each sublayer list. However, we note that, when $k \geqslant 10$, the PL Index begins to show much better performance than the other methods because the PL Index reads at most one sublayer from a specific sublayer list when finding top-i ($2 \leqslant i \leqslant k$). In addition, *No_Objects_Read* increases approximately linearly as k increases because the sublayer sizes of the PL Index are similar to one another. The PL Index improves by 0.1–2.2 times over CSI_ONION, by 0.8–8.8 times over ONION, by 0.6–7.3 times over PREFER, 0.2–2.9 times over AppRI, and by 0.2–4.6 times over LPTA. Fig. 18a also shows that except the PL Index, CSI_ONION outperforms ONION and other existing methods. This is because the layer sizes of CSI_ONION are much smaller than those of ONION. Hereafter, we exclude ONION and PREFER from the comparison of the query performance because AppRI and LPTA, being more recent work, significantly outperform ONION and PREFER [10,22] as shown in Fig. 18a.

Experiment 8: query performance as N is varied

Fig. 18b shows the query performance of the PL Index, CSI_ONION, AppRI, and LPTA as N is varied from 1 K to 100 K. Here, we have different optimal partition levels for different data sizes (i.e., we use $h_o = 2$ when $N = 1$ K and $h_o = 3$ when $N = 10$ K and 100 K). Fig. 18b shows that the PL Index improves by 1.5–1.9 times over CSI_ONION, by 1.9–3.8 times over AppRI, and by 2.2–3.8 times over LPTA.

Experiment 9: query performance as d is varied

Fig. 18c shows the query performance as d is varied from 2 to 7. Here, the optimal partition level is different for each dimension (i.e., we use $h_o = 4$ when $d = 2$, $h_o = 3$ when $d = 3$, $h_o = 2$ when $d = 4$ and 5, and $h_o = 1$ when d = 6 and 7). Fig. 18c shows that the PL Index improves by 1.3–1.9 times over CSI_ONION, by 2.4–2.7 times over AppRI, and of 1.6–5.1 times over LPTA.

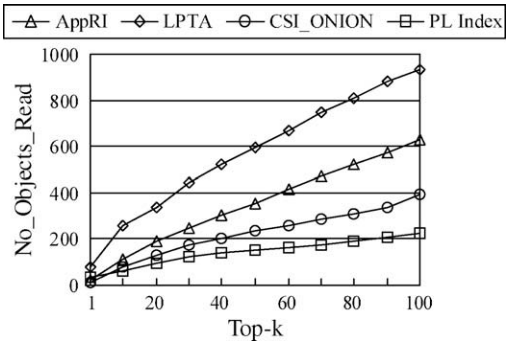


Fig. 20. The comparison of the query performance as k is varied (Cover3D, $N = 10$K and $d = 3$).

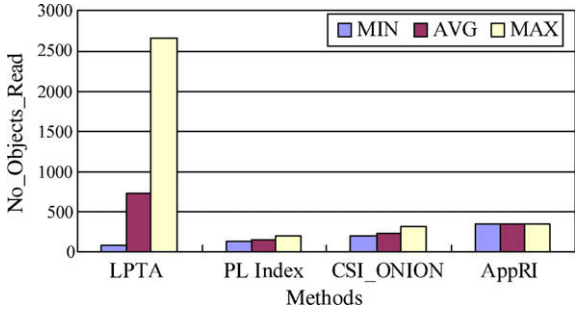


Fig. 21. The comparison of the sensitivity of the query performance to the preference vector (Cover3D, $N = 10$ K, $d = 3$, and $k = 50$).

Experiment 10: sensitivity of the query performance to the preference vector
Fig. 19 shows the sensitivity of the query performance to the preference vector. As mentioned in Section 4.3, the query performance of the PL Index is expected to be rather insensitive to the preference vector because the sublayer sizes of the PL Index are similar to one another. Fig. 19 shows that the query performances of CSI_ONION, ONION, LPTA, and PREFER are much affected by the preference vector [22], but those of AppRI and the PL Index are quite insensitive [22].

Experiment 11: query performance as k is varied when using a real data set
Fig. 20 shows the query performance of the PL Index, CSI_ONION, AppRI, and LPTA as k is varied from 1 to 100 when using Cover3D, a three-dimensional real data set. Here, we use $h_o = 2$, which is the optimal partition level when $k = 50$. Fig. 20 shows the PL Index begins to outperform the other methods when $k \geqslant 8$. This tendency is similar to that of the synthetic data set shown in Fig. 18a. The PL Index improves the performance by 0.3–1.7 times over CSI_ONION, by 0.7–2.8 times over AppRI, and by 2.3–4.3 times over LPTA. The PL Index is insensitive to the preference vector for the Cover3D data set as shown in Fig. 21. The index building times of the PL Index is less than those of CSI_ONION and AppRI(naive). This tendency is similar to that of the synthetic data set as shown in Fig. 16b when $d = 3$. The index building times of the PL Index (1246 ms) improves by 3.1 times over CSI_ONION and by 112.1 times over AppRI(naive).

8. Conclusions

In this paper, we have proposed the PL Index that significantly improves the top-k query processing performance. We have also proposed the DynamicLayerOrdering algorithm that effectively evaluates top-k queries by using the PL Index. The PL Index partitions the universe (the database) into a number of subregions, and then, constructs a sublayer list for each subregion.

Building the PL Index consists of two steps: the partitioning step and the layering step. For the partitioning step, we have proposed a partitioning algorithm that makes the sizes of sublayers within a sublayer list as uniform as possible taking advantage of the assumption that queries have monotone linear score functions. For the layering step, we have formally defin ed a concept of the layer called the convex skyline. We have shown that the sublayer list constructed using the convex skylines satisfies the optimally linearly ordered property (Lemma 5). We have proved using this property that the DynamicLayerOrdering algorithm finds a top-i ($2 \leqslant i \leqslant k$) object by reading *at most one* sublayer from a specific sublayer list (Theorem 1). Finally, we have analyzed the time complexity of the PL Index building algorithm and DynamicLayerOrdering algorithm.

We have performed extensive experiments on synthetic and real data sets by varying the data size N, the dimension d, the number of results k, and the preference vector p. Experimental results show that the PL Index significantly outperforms existing methods except for small values of k in query performance (say, $k \leqslant 9$ for UNIFORM data and $k \leqslant 7$ for Cover3D real data). They also show that the query performance of the PL Index is quite insensitive to the preference vector and is approximately linear in the value of k (due to Theorem 1). The results using a real data set (Cover3D) show a tendency similar to those using synthetic data sets.

Acknowledgements

This work was partially supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korean Government (MEST) (No. R0A-2007-000-20101-0). This work was also partially supported by the Internet Services Theme Program funded by Microsoft Research Asia.

Appendix A. Functions for the universepartitioning algorithm

See Fig. 22.

Appendix B. Proof of Lemma 2

As defined in Table 1, the coordinate values of $vMAX^+$ are larger than those of all the objects in S. f is monotone. Thus, $\forall t \in S, f(vMAX^+) > f(t)$. □

Appendix C. Proof of Lemma 3

For an arbitrary monotone linear score function f, $\exists t \in SL(S)$ such that t minimizes f over the objects in S according to the definition of the skyline [6]. Let $T = SL(S) \cup \{vMAX^+\}$. Then, from the properties of the convex hull, $\exists t \in CH(T)$ such that t minimizes f over the objects in T [12], and accordingly, over the objects in $S \cup \{vMAX^+\}$. Since $vMAX^+$ cannot minimize f by Corollary 1, at least one object that minimizes f over the objects in S must exist in $CH(T) - \{vMAX^+\}$. □

Function ***DistributeObjects*****( )**

Input: (1) S: a set of d-dimensional objects
(2) o: the origin of the new coordinates that optimally partitions the region containing S

Output: $\{S_1, S_2, \ldots, S_d\}$: a set of d disjoint subsets of S

1. FOR $j := 1$ to d DO $T_j := \{\}$ /* initialization */
2. FOR $t \in S$ DO /* distributing objects */
   2.1 Find j such that $t[j] - o[j] = min_{i=1}^{d}(t[i] - o[i])$
   /* find the farthest axis X_j for $t' = t - o$ in the new coordinates */
   2.2 $T_j := T_j \cup \{t\}$
3. RETURN $\{T_1, T_2, \ldots, T_d\}$

Function ***FindOptimalOffset*****( )**

Input: (1) x: the index of the axis X_x examined
(2) S: a set of d-dimensional objects
(3) o: the origin of the new coordinates
(4) w: the width of the region containing S

Output: δ: the offset at which objects are optimally distributed

1. $o' := o$ /* initialization */
2. FOR $k := 1$ to m DO /* m is the number of points examined */
   2.1 $o'[x] := o'[x] - w[x]/m$ /* shifting o' along the negative direction of the axis X_x by $w[x]/m$ */
   2.2 $\{T_1, \ldots, T_d\} := DistributeObjects(S, o')$
   2.3 $SD[k] :=$ the standard deviation of $\{|T_1|, |T_2|, \ldots, |T_d|\}$
3. Find u such that $SD[u] = min_{k=1}^{m}(SD[k])$
4. RETURN $\delta := u*w[x]/m$

Fig. 22. Functions for the UniversePartitioning algorithm.

Appendix D. Proof of Lemma 4

Let us assume that $\exists t \in CS(S)$ such that t only maximizes f over the objects in $SL(S)$. Then, since $CS(S) = CH(T) - \{vMAX^+\}$, where $T = SL(S) \cup \{vMAX^+\}$, $\exists t \in CH(T)$ such that t only maximizes f over the objects in $SL(S)$ and $t \neq vMAX^+ \notin SL(S)$. However, since only $vMAX^+$ maximizes f over the objects in T by Lemma 2, any object that only maximizes f over the objects in $SL(S)$ cannot be a vertex of the convex hull of T. Thus, such an object cannot exist in $CH(T)$. This contradicts the assumption. □

Appendix E. Proof of Lemma 5

Let $L = L[1]L[2], \ldots, L[m]$ be the list of convex skylines constructed over a set S of objects. Let us assume that L is not an optimally linearly ordered set. Then, for some monotone linear score function $f(t)$ and two convex skylines $L[i]$, $L[j] (1 \leqslant i < j \leqslant m)$, $\exists t_q (f(t_p) > f(t_q))$, where $t_q \in L[j]$ and t_p is one of the objects that have the lowest score in $L[i]$ for $f(t)$. According to Lemma 3, when constructing $L[1]$ through $L[i]$, one such t_q must have been included in $L[1]$ or … or $L[i]$. This contradicts the assumption. □

References

[1] B. Barber, D. Dobkin, H. Huhdanpaa, The quickhull algorithm for convex hulls, ACM Trans. Math. Software 22 (4) (1996) 469–483.
[2] M. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum, IO-Top-k: index-access optimized top-k query processing, in: Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB), Seoul, Korea, Sept. 2006, pp. 475–486
[3] J.L. Bentley, H.T. Kung, M. Schkolnick, C.D. Thompson, On the average number of maxima in a set of vectors and applications, J. ACM 33 (2) (1978) 536–543.
[4] M. Berg, M. Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry: Algorithms and Applications, second ed., Springer-Verlag, 2000.
[5] G. Beskales, M.A. Soliman, I.F. Ilyas, Efficient search for the top-k probable nearest neighbors in uncertain databases, in: Proceedings of the 34th International Conference on Very Large Data Bases (VLDB), Auckland, New Zealand, Aug. 2008, pp. 326–339.
[6] S. Borzsonyi, D. Kossmann, K. Stocker, The skyline operator, in: Proceedings of the 17th International Conference on Data Engineering (ICDE), Heidelberg, Germany, Apr. 2001, pp. 421–430.

[7] C. Buchta, On the average number of maxima in a set of vectors, Informat. Process. Lett. 33 (4) (1989) 63–65.
[8] Y.C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, J.R. Smith, The onion technique: indexing for linear optimization queries, in: Proceedings of the International Conference on Management of Data, ACM SIGMOD, Dallas, Texas, May 2000, pp. 391–402.
[9] C.-Y. Chan, P.-K. Eng, K.-L. Tan, Stratified computation of skylines with partially-ordered domains, in: SIGMOD, 2005.
[10] G. Das, D. Gunopulos, N. Koudas, D. Tsirogiannis, Answering top-k queries using views, in: Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB), Seoul, Korea, Sept. 2006, pp. 451–462.
[11] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, in: Proceedings of the 20th ACM Symposium on Principles of Database Systems (PODS), Santa Barbara, California, May 2001, pp. 102–113.
[12] S.G. Gass, Linear Programming: Method and Applications, fifth ed., An International Thomson Publishing Company, 1985, 1985.
[13] G. Hadley, Linear Programming, Addison-Wesley Publishing Company, 1962.
[14] E. Horowitz, S. Sahni, S.A. Freed, Fundamentals of Data Structures in C, Computer Science Press, 1993.
[15] V. Hristidis, Y. Papakonstantinou, Algorithms and applications for answering ranked queries using ranked views, VLDB J. 13 (1) (2004).
[16] M. Hua, J. Pei, W. Zhang, X. Lin, Ranking queries on uncertain data: a probabilistic threshold approach, in: Proceedings of the International Conference on Management of Data, ACM SIGMOD, Vancouver, Canada, June 2008, pp. 673–686.
[17] W. Jin, A.K.H. Tung, M. Ester, J. Han, On efficient processing of subspace skyline queries on high dimensional data, in: Proceedings of the 19th International Conference on Science and Statistical Database Management (SSDBM), Banff, Canada, July 2007, p. 12.
[18] C. Li, K.C.-C. Chang, I.F. Ilyas, S. Song, RankSQL: query algebra and optimization for relational top-k queries, in: Proceedings of the International Conference on Management of Data, ACM SIGMOD, Baltimore, Maryland, June 2005, pp. 131–142.
[19] D. Papadias, Y. Tao, G. Fu, B. Seeger, Progressive skyline computation in database systems, ACM Trans. Database Syst. 30 (1) (2005).
[20] K.-L. Tan, P.-K. Eng, B.C. Ooi, Efficient progressive skyline computation, in: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), San Francisco, CA, Sept. 2001, pp. 301–310.
[21] K. Yi, H. Yu, J. Yang, G. Xia, Y. Chen, Efficient maintenance of materialized top-k views, in: Proceedings of the 19th International Conference on Data Engineering (ICDE), Bangalore, India, Mar. 2003, pp. 189–200.
[22] D. Xin, C. Chen, J. Han, Towards robust indexing for ranked queries, in: Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB), Seoul, Korea, Sept. 2006, pp. 235–246.
[23] L.A. Zadeh, Toward a generalized theory of uncertainty (GTU), Informat. Sci. 172 (1–2) (2005) 1–40.
[24] L.A. Zadeh, Is there a need for fuzzy logic?, Informat Sci. 178 (13) (2008) 2751–2779.