The Hybrid-Layer Index: A Synergic Approach to Answering Top-*k* Queries in Arbitrary Subspaces

Jun-Seok Heo [†], Junghoo Cho [‡], and Kyu-Young Whang [†]

[†]*Department of Computer Science, KAIST, Korea,* {jsheo,kywhang}@mozart.kaist.ac.kr
[‡]*University of California, Los Angeles, USA,* cho@cs.ucla.edu

***Abstract*— In this paper, we propose the *Hybrid-Layer Index* (simply, the *HL-index*) that is designed to answer top-k queries efficiently when the queries are expressed on any *arbitrary subset* of attributes in the database. Compared to existing approaches, the HL-index significantly reduces the number of tuples accessed during query processing by pruning unnecessary tuples based on two criteria, i.e., it filters out tuples both (1) *globally* based on the combination of *all* attribute values of the tuples like in the layer-based approach (simply, *layer-level filtering*) and (2) based on *individual* attribute values specifically used for ranking the tuples like in the list-based approach (simply, *list-level filtering*). Specifically, the HL-index exploits the synergic effect of integrating the layer-level filtering method and the list-level filtering method. Through an in-depth analysis of the interaction of the two filtering methods, we derive a tight bound that reduces the number of tuples retrieved during query processing while guaranteeing the correct query results. We propose the HL-index construction and retrieval algorithms and formally prove their correctness. Finally, we present the experimental results on synthetic and real datasets comparing the performance of the HL-index to other state-of-the-art indexes. Our experiments demonstrate that the HL-index shows the best (or close to best) performance in most scenarios regardless of the size of the dataset, the number of attributes in the tuples, and the number of attributes used in the queries.**

I. Introduction

Computing top-k answers quickly is becoming ever more important as the size of databases grows and as more users access data through interactive interfaces. When a database is large, it may take minutes (if not hours) to compute the complete answer to a query if the query matches millions of the tuples in the database. Most users, however, are interested in looking at just the top few results (ranked by a small set of attribute values that the users are interested in) and they want to see the results immediately after they issue the query.

As an example, consider a database of digital cameras, which has many attributes such as price, manufacturer, model number, weight, size, pixel count, sensor size, etc. Among these attributes, a particular user is likely to be interested in a small subset when they make a decision to purchase. For example, a user who wants to buy a cheap compact digital camera will be mainly interested in the price and the weight and may issue a query like

```
SELECT * FROM Cameras
ORDER BY 0.5*price+0.5*weight ASC LIMIT k.
```

Another user who primarily cares about the quality of the pictures will be more interested in the pixel count and sensor size and issue a query like

```
SELECT * FROM Cameras
ORDER BY 0.4*pixelCount+0.6*sensorSize DESC
LIMIT k.
```

To handle scenarios like the above, we propose the *Hybrid-Layer Index* (simply, the *HL-index*) that is designed to answer top-k queries on an *arbitrary subset* of the attributes efficiently. There exist a number of approaches for efficient computation of top-k answers. For example, in their seminal work, Fagin et al.[10], [11] designed a series of algorithms that consider a tuple as a potential top-k answer only if the tuple is ranked high in *at least* one of the attributes used for ranking. We refer to this approach as the *list-based approach* because the algorithms require maintaining one sorted list per each attribute. While this approach shows significant improvement compared to earlier work, it often considers 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 the list-based approach has to consider it because of its high rank in that one attribute. As the size of the database grows, this becomes an acute problem because there are likely to be more tuples that are ranked high in one attribute but low overall.

To avoid this pitfall, Chang et al.[7] proposed an algorithm that constructs a global index based on the combination of *all* attribute values and uses this index for top-k answer computation. We refer to this approach as the *layer-based approach* because it builds an index that partitions the tuples into multiple layers. The layer-based approach avoids the pitfall of the Fagin's algorithms, but it also has the opposite problem. Because the index is constructed on *all* attributes, it does not perform well when the query ranks tuples by a small *subset* of the attributes. A tuple may be ranked high globally on many attributes, but it may be ranked low for a particular subset of attributes used for a query.

One simple way to address the drawback of the layer-based approach is to build one dedicated index per *every subset* of attributes and use the appropriate index for a query as in [9], [14]. We refer to these approaches as the *view-based approach*. Clearly, view-based approaches lead to high query performance, but they also incur significant space overhead.

Our proposed HL-index tries to avoid all pitfalls of the existing approaches in the following ways. By careful integration of the list-based and the layer-based approaches, it is able to filter out a tuple both by the *global* combination of *all* of its attribute values (like in the layer-based approach) and by the *individual* consideration of the particular attribute values used for ranking (like in the list-based approach). In addition, one HL-index can handle *any* queries on an *arbitrary subset* of the attributes avoiding the space overhead of the view-based approach. More precisely, we make the following contributions in this paper.

- We propose the HL-index that can be used for answering top-k queries on an arbitrary subset of attributes. The HL-index can be built for either (1) linear scoring functions (including monotone and non-monotone linear functions) or (2) monotone scoring functions (including linear and non-linear monotone functions). The HL-index has significantly more pruning power than existing approaches and does not require a separate index customized for each class of queries on different subsets of attributes.
- We present the algorithms for processing top-k queries using the HL-index. Through an in-depth analysis of the interaction of the list-based and layer-based approaches, we derive a tight bound to minimize the number of tuples that are retrieved during query processing and to guarantee the correctness of the computed results. We also provide formal proofs of correctness of those algorithms.
- We conduct extensive experiments comparing the performance of the HL-index with those of existing approaches on both synthetic and real data. The HL-index can exploit the synergic effect of the list-based approach and the layer-based approach by meticulous integration of the two approaches. As a result, the HL-index shows better performance over existing approaches for practically all settings in our experiments. In particular, our experiments show that the HL-index performs particularly well when the size of the database is large, leading to a factor of three or more improvement for a database of million tuples in our experiments.

The rest of the paper is organized as follows: We first go over related work in Section II and we formally define the top-k queries that we handle in Section III. Then, in Section IV, we describe the HL-index construction algorithm and, in Section V, explain the top-k query processing algorithm using the HL-index and prove its correctness. In Section VI we present our experiments that compare the performance of the HL-index to existing approaches. We conclude the paper in Section VII.

II. Related Work

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

A. Layer-Based Approach

The layer-based approach 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 i^{th} 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. ONION [7] and AppRI [21] are well-known methods of this approach.

ONION [7] 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 *the optimally linearly ordered set*, Chang et al. [7] has shown that ONION answers top-k queries by reading at most k layers starting from the outmost layer.

ONION is capable of answering a query with an arbitrary (monotone or non-monotone) linear function because of the geometrical properties of the convex hull [12]. On the other hand, the query performance is sometimes adversely affected due to the relatively large sizes of layers [21], particularly when the number of attributes mentioned in the query is small, because it reads all the tuples in a layer. AppRI [21] constructs a list of layers as well, but exploiting the domination relation of skylines.

B. List-Based Approach

The list-based approach 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 well-known method of the list-based approach, sequentially accesses each sorted list mentioned in a query in parallel. That is, for all attributes appearing in a query, it accesses the first element of each sorted list, then the second 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.

Under the list-based approach, since the lists are independent of one another, top-k tuples are computed by accessing only those lists corresponding to the attributes mentioned in the query. That is, it can filter out unnecessary tuples by individual consideration of these attribute values. However, since TA does not exploit the relationship among the attributes when creating the sorted lists, its performance gets worse as the number of attributes mentioned in the query increases.

C. View-Based Approach

The basic idea behind the view-based approach is to "pre-compute" the answers to a class of queries on every subset of attributes 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 [14] and LPTA [9] are well-known methods of this approach. Because the view-based approach requires constructing an index for each subset of attributes, its space and maintenance overhead often becomes an important issue in using this approach for a practical system [20].

D. Other Approaches

There exists a large body of work for efficient computation of skyline queries [6], [17], [18]. Because the skyline contains at least one tuple that minimizes any monotone scoring function [5], this body of work can be used to deal with top-k queries under a monotone scoring function for the special case of $k = 1$. SUB-TOPK [19] is one extension of these methods that finds the top-k results for any k value, but because it is still based on the skyline approach, it only deals with monotone scoring functions and is unable to handle non-monotone linear functions.

Table I summarizes the top-k indexing methods that are compared in this paper and the functions they support.

TABLE I
TOP-k INDEXING METHODS COMPARED AND THE FUNCTIONS THEY SUPPORT.

Functions	Linear	Non-linear
Monotone	TA, ONION, AppRI, PREFER, LPTA, SUB-TOPK	TA, PREFER, SUB-TOPK
Non-monotone	ONION	

III. PROBLEM DEFINITION

In this section, we formally define the problem of top-k queries when the tuples are ranked by an arbitrary subset of attributes. A target relation R of N tuples has d attributes $A_1, A_2, \ldots, A_d$. The value of each attribute A_i is assumed to range between $[0.0, 1.0]$, so 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. A scoring function $f(t) : t \to [-1.0, 1.0]$ maps each object $t \in [0.0, 1.0]^d$ to a real value in $[-1.0, 1.0]$. Then, a top-k query is to find the k objects in R that have the lowest (or highest) score under $f(t)$. 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 $f(t^1) \le f(t^2) \le \ldots \le f(t^k) \le f(t^l)$, $k+1 \le l \le N$. Here, t^j denotes the j^{th} ranked object in the ascending order of their score, where $1 \le j \le N$.

The scoring function for top-k queries is generally assumed to be either linear [7], [9], [14], [21] or monotone [9], [11], [14], [19], [21]. A *linear* scoring function is a function of the following form

$$f_{\bar{w}}(t) = \sum_{i=1}^{d} w[i] * t[i] \tag{1}$$

where $t[i]$ is the i^{th} attribute value of t and $w[i]$ is the "weight" of the i^{th} attribute. The vector of $w[i]$ values, $\bar{w}$, is referred to as the *preference vector*. Without loss of generality, the $w[i]$ values are assumed to range between $[-1.0, 1.0]$ and are normalized to be $\sum_{i=1}^{d} |w[i]| = 1$. A *monotone* scoring function satisfies the following condition [11]:

$$\text{If } t[i] \le t'[i] \text{ for all } i = 1, \ldots, d, \text{ then } f(t) \le f(t').$$

Informally, monotony means that if an object has smaller scores than others in *all* attributes, then its overall score should also be smaller. We note that a linear function $f_{\bar{w}}$ is monotone if and only if its $w[i]$ values are all non-negative. Depending on the sign of the $w[i]$ values, a linear function may be non-monotone.

As we will explain, our HL-index can be designed to deal with *either* of the following classes: (1) *all* linear functions including monotone and non-monotone linear functions; (2) *all* monotone functions including linear and non-linear monotone functions. Due to space limit and for the clarify of our exposition, however, we mainly assume linear scoring functions (monotone or non-monotone) in the rest of this paper and briefly deal with the variation for non-linear monotone functions in Section V-E.

TABLE II
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 i^{th} attribute of R $(1 \le i \le d)$
t	an object in R (t is considered as a d-dimensional vector $\bar{t}$ that has $t[i]$ as the i^{th} element)
$t[i]$	the value of attribute A_i in the object t $(t \in R)$
$\bar{w}$	a preference vector (a d-dimensional vector that has $w[i]$ as the i^{th} element)
$w[i]$	the weight of attribute A_i in the preference vector $\bar{w}$
L_i	the i^{th} layer or the set of objects in the i^{th} layer
$L_{i,j}$	the list of objects in L_i sorted in the ascending order of their A_j values
$o_{min}(S)$	the minimum-scored object in the set S
$H(i)$	$\{o \mid f_{\bar{w}}(o) \le f_{\bar{w}}(o_{min}(L_i))$ for $o \in L_1 \cup L_2 \cup \cdots \cup L_i\}$
$S_{i,j}(n)$	the set of the first n objects from the head (or tail) of $L_{i,j}$
$S_i(n)$	$S_{i,1}(n) \cup S_{i,2}(n) \cup \cdots \cup S_{i,d}(n)$
$U_i(n)$	$L_i - S_i(n)$; the objects in L_i that are not in $S_i(n)$
$a_{i,j}(n)$	the A_j value of the n^{th} object from the head (or tail) of $L_{i,j}$
$\mathcal{F}_i(n)$	$f_{\bar{w}}(a_{i,1}(n), a_{i,2}(n), \ldots, a_{i,d}(n))$; no object in $U_i(n)$ has a score lower than $\mathcal{F}_i(n)$.

As we stated in Introduction, when R has many attributes, any particular top-k query is likely to have nonzero weights only for a small subset of the attributes [19]. To emphasize this fact, we use *SUB* to denote the set of attributes with $w[i] \ne 0$ and call the size of *SUB* the *sub-dimension* and the space consisting of these attributes the *subspace*. That is, *SUB* $= \{i \mid w[i] \ne 0 \text{ for } i = 1, \ldots, d\}$. Under this notation, a subspace top-k query is to find the k lowest-scored objects $[t^1, \ldots, t^k]$ given the query triple $(\mathit{SUB}, f_{\bar{w}}(), k)$. In Table II, we summarize the notation that we use throughout this paper. The symbols that have not been introduced yet will be explained in Section V.

IV. Hybrid-Layer Index (HL-index)

We now explain how to construct an HL-index to efficiently handle subspace top-k queries. The primary goal of the HL-index is to enable both *layer-level filtering* and *list-level filtering*: (1) The layer-level filtering prunes an object by the *global* combination of *all* of its attribute values like in the layer-based approach. (2) The list-level filtering prunes an object by the *individual* consideration of the particular attribute values with nonzero weights in the scoring function like in the list-based approach.

To enable the two types of filtering, an HL-index is constructed in two steps: (1) *Layering step*: In this step, objects in the relation R are partitioned into a disjoint set of layers, $\{L_1, L_2, \ldots, L_m\}$, where L_i represents the i^{th} layer. Every object belongs to one and only one layer. As we will see later, once the objects are partitioned into layers, the top-k objects can be obtained from *at most* the first k layers; objects in all the other layers can be ignored, enabling the layer-level filtering. (2) *Listing step*: In this step, for each layer L_i, we construct d sorted lists $\{L_{i,1}, L_{i,2}, \ldots, L_{i,d}\}$, where $L_{i,j}$ represents the list of objects in L_i sorted in the ascending order of their j^{th} attribute values.

As we mentioned earlier, our HL-index can be built for either all linear functions or for all monotone functions. Figure 1 describes the version of the HL-index construction algorithm for all linear functions.[1] The input to the algorithm is the set R of the d-dimensional objects, and the output is the set of layers $L = \{L_1, L_2, \ldots, L_m\}$, where the i^{th} layer L_i contains d sorted lists, $L_i = \{L_{i,1}, \ldots, L_{i,d}\}$. We explain the algorithm using Example 1.

```
Algorithm LayerbasedListBuilding:
Input : R: a set of d-dimensional objects
Output: L: the list of the sets of d sorted lists (the HL Index)
Algorithm:
1. WHILE ($R \neq \{\}$) DO BEGIN
2.     $i := i + 1$   /* layer number, i is initialized with 0 */
3.     $R_i$ := objects at the vertices of the convex hull over R  /* i-th layer */
4.     FOR $j := 1$ to d DO   /* for each attribute */
5.         Sort the objects in $R_i$ in the ascending order of their j-th attribute
           values and store their identifiers as the list $L_{ij}$
6.     $L_i := \{L_{i,1}, \ldots, L_{i,d}\}$   /* the set of d sorted lists */
7.     $R := R - R_i$
8. END /* WHILE */
9. RETURN $L := [L_1 = \{L_{1,1}, \ldots, L_{1,d}\}, \ldots, L_m = \{L_{m,1}, \ldots, L_{m,d}\}]$
```

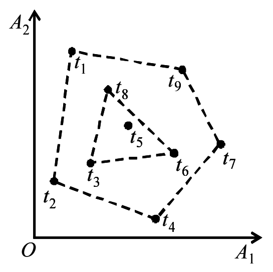
Fig. 1. The LayerbasedListBuilding algorithm for building the HL-index.

Example 1: Let us assume that the input relation R has nine objects, $t_1, \ldots, t_9$, and two attributes, A_1 and A_2, as we show in Figure 2(a). Here, *ID* represents the identifier of the object. Given this input relation, in Line 3, the algorithm finds the convex hull and places the objects at the vertices of this convex hull, $\{t_1, t_2, t_4, t_7, t_9\}$, into R_1 (R_i is the set that contains all objects in the i^{th} layer L_i) as shown in the left-most rectangle in Figure 2(c). Why we use the convex hull to partition R will be explained later in Section V. Then, in Lines 4 and 5, the algorithm constructs two sorted lists, $L_{1,1}$ and $L_{1,2}$, for the five objects in R_1. More specifically, $L_{1,1}$ and $L_{1,2}$ list the object IDs in R_1 in the ascending order of their A_1 and A_2 values, respectively. For example, in Figure 2(d) the first object in $L_{1,1}$ is t_2 because t_2 is the object with the smallest A_1 value in R_1. As the algorithm proceeds, it iteratively constructs a new convex hull with the remaining objects until the input set R becomes empty. Eventually, the algorithm constructs three layers, L_1, L_2 and L_3, for the input relation R, as shown in Figure 2(d).

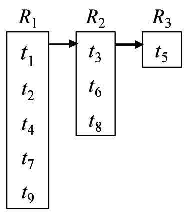
[1]Again, the HL-index for all monotone functions are described briefly in Section V-E.

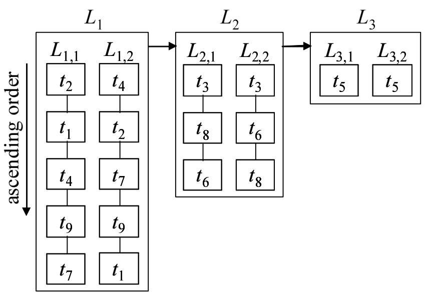
ID	A_1	A_2
t_1	0.2	1.0
t_2	0.1	0.3
t_3	0.3	0.4
t_4	0.65	0.1
t_5	0.5	0.6
t_6	0.75	0.45
t_7	1.0	0.5
t_8	0.4	0.8
t_9	0.8	0.9

(a) A target relation R.



(b) The convex hull vertices over the objects in R.





(c) The layer list. (d) The HL-index.

Fig. 2. An example of constructing the layer lists and the HL-index in the two-dimensional universe.

Before we proceed, we emphasize that, in our HL-index $L = \{L_1 = \{L_{1,1}, \ldots, L_{1,d}\}, \ldots, L_m = \{L_{m,1}, \ldots, L_{m,d}\}\}$, each sorted list $L_{i,j}$ contains only object identifiers, but not the full attribute values of the objects. We store their full attribute values in a separate place. Therefore, once we obtain an object ID from the HL-index, we will have to retrieve the full values of its attributes separately in order to compute the object score under a given scoring function as done by Chaudhuri et al. [8].

V. Query Processing Using the HL-index

We now discuss how we can use the HL-index for exploiting the synergic effect of layer-level filtering and list-level filtering in computing the top-k objects. In Section V-A, we start reviewing the ONION algorithm [7] to explain how the HL-index can be used for layer-level filtering. Then in Section V-B, we explain how we can extend the ONION algorithm to enable list-level filtering.

A. ONION Algorithm: Layer-Level Filtering

In the HL-index construction algorithm of Figure 1, the input objects in R are partitioned into multiple layers by the repeated extraction of the convex hull vertices. This layering strategy was proposed by Chang et al. [7], where the authors proved that the top-k objects are guaranteed to be in the first k layers L_1 through L_k. Therefore, in computing the top-k answers, all objects in the layer L_{k+1} and above can be ignored, making layer-level filtering possible. More precisely, Chang et al. [7] proved the following important theorem.

Theorem 1: [7] **(Optimally Linearly Ordered Set Property)** Let $L = \{L_1, L_2, \dots, L_m\}$ be the set of layers constructed by the recursive extraction of convex hull vertices. Let $o_{min}(L_i)$ be the minimum-scored object in L_i. Then, no object in the layers $L_{i+1}, \dots, L_m$ can have a score less than $o_{min}(L_i)$. That is, $\forall o \in L_j \, (i < j \le m), f_{\bar{w}}(o_{min}(L_i)) \le f_{\bar{w}}(o)$ under any preference vector $\bar{w}$.

Theorem 1 implies that if we have found k or more objects whose scores are lower than or equal to $f_{\bar{w}}(o_{min}(L_i))$ from the layers L_1 through L_i, then we can ignore all objects in L_{i+1} through L_m. More precisely, Chang et al. [7] proved the following corollary.

Corollary 1: [7] Let $o_{min}(L_i)$ be the minimum-scored object in the layer L_i. Let $H(i)$ be the objects in $L_1, L_2, \dots, L_i$ whose scores are $f_{\bar{w}}(o_{min}(L_i))$ or less. That is, $H(i) = \{o | f_{\bar{w}}(o) \le f_{\bar{w}}(o_{min}(L_i))$ for $o \in L_1 \cup L_2 \cup \dots \cup L_i\}$.[2] If $H(i)$ contains k or more objects (i.e., if $|H(i)| \ge k$), $H(i)$ contains the top k objects.

Based on Corollary 1, Chang et al. [7] proposed the ONION algorithm. In Figure 3 we show a slightly modified version of the ONION algorithm.[3] Starting from $i = 1$, the ONION algorithm retrieves *all* objects in L_i and evaluates their scores in Line 2. Once all object scores are evaluated, it identifies $o_{min}(L_i)$, the minimum-scored object in L_i, in Line 3 and computes $H(i)$, the set of objects in L_1 through L_i with scores $f_{\bar{w}}(o_{min}(L_i))$ or less, in Line 4. Then, in Lines 5 and 6, the algorithm returns the top-k objects from $H(i)$ if $H(i)$ contains k or more objects. Otherwise, it increases i by one and repeats the process.

```
1. FOR $i = 1$ to $m$ DO BEGIN
2.     Evaluate $f_{\overline{w}}(o)$ for all $o \in L_i$
3.     Find $o_{min}(L_i)$ from $L_i$ where $f_{\overline{w}}(o_{min}(L_i)) \le f_{\overline{w}}(o)$ for any $o \in L_i$
4.     Compute $H(i) = \{o \mid f_{\overline{w}}(o) \le f_{\overline{w}}(o_{min}(L_i))$ for $o \in L_1 \cup \dots \cup L_i\}$
5.     IF $|H(i)| \ge k$ THEN
6.         RETURN the $k$ objects in $H(i)$ with the lowest scores
7. END /* FOR */
```

Fig. 3. The modified ONION algorithm.

We note that the ONION algorithm performs the layer-level filtering by retrieving objects only from the first few layers. In particular, the i value in the algorithm never increases beyond k, so even in the worst case scenario, at most the first k layers are retrieved [7]. However, the main drawback of the ONION algorithm is that *all* objects in these layers, which could be large, have to be retrieved to evaluate their scores. In the next section, we explain how we can use the individual lists in the HL-index to perform list-level filtering by retrieving only a subset of objects in each layer.

[2]In the original definition, the inequality sign should have been "$\ge$" instead of "$>$" [15], and for finding k objects with the lowest scores as the results, we use "$\le$".

[3]We present a slightly modified algorithm from what was proposed by Chang et al. [7] in order to make our later discussion easier to follow.

B. List-Level Filtering For HL-Index

In designing the algorithm that retrieves only a subset of objects from each layer, we first note that the only reason why the ONION algorithm retrieves all objects from L_i in Line 2 is to be able to identify $o_{min}(L_i)$ and $H(i)$ in Lines 3 and 4. In other words, *as long as we can identify $o_{min}(L_i)$ and $H(i)$ correctly, we do not have to retrieve all objects in L_i.* The main challenge for allowing the list-level filtering is then to figure out the way to identify $o_{min}(L_i)$ and $H(i)$ without evaluating the score $f_{\bar{w}}$ for every object in L_i.

To explain how we can achieve this using the HL-index, we first introduce relevant notation. Here, for ease of understanding, we consider the notation to be used for handling monotone linear functions and extend it to handle non-monotone linear functions in Section V-C.1. We use $S_{i,j}(n)$ to refer to the set of the first n objects at the head of the list $L_{i,j}$. For example, $S_{1,2}(3) = \{t_2, t_4, t_7\}$ in Figure 2(d). We set $S_i(n)$ to be $S_{i,1}(n) \cup S_{i,2}(n) \cup \dots \cup S_{i,d}(n)$. Informally, $S_i(n)$ can be considered as the set of objects that we "see" by retrieving the first n objects from each list $L_{i,j}$. For instance, $S_1(2) = \{t_1, t_2, t_4\}$ in Figure 2(d). We set $U_i(n) = L_i - S_i(n)$. For example, $U_1(2) = L_1 - S_1(2) = \{t_1, t_2, t_4, t_7, t_9\} - \{t_1, t_2, t_4\} = \{t_7, t_9\}$. Informally, $U_i(n)$ can be considered as the set of the objects in L_i that are not "seen" by retrieving the top n objects from each list $L_{i,j}$. We use $a_{i,j}(n)$ to refer to the A_j attribute value of the n^{th} object at the head of the list $L_{i,j}$. For example, in Figure 2(d), $a_{1,2}(3)$ is 0.5, the A_2 value of the third object t_7 in $L_{1,2}$. Since each list $L_{i,j}$ is sorted by the A_j values, $a_{i,j}(n)$ monotonously increases as n increases. Finally, we set $\mathcal{F}_i(n) = f(a_{i,1}(n), a_{i,2}(n), \dots a_{i,d}(n))$. The meaning of the new set of symbols is summarized in Table II. Under this notation, Fagin et al. [11] proved the following important theorem:

Theorem 2: [11] Under any monotone (linear or non-linear) function f, every object in $U_i(n)$ has a score larger than or equal to the threshold value $\mathcal{F}_i(n)$. That is, $\forall o \in U_i(n),\ f(o) \ge f\,(a_{i,1}(n), a_{i,2}(n), \dots, a_{i,d}(n)) = \mathcal{F}_i(n)$.

1) Identifying $o_{min}(L_i)$: Theorem 2 provides an important clue on how we can identify the $o_{min}(L_i)$ from L_i without retrieving all objects in L_i. In particular, under a monotone (linear or non-linear) scoring function f, the theorem guarantees that after we retrieve $S_i(n)$, the first n objects from each list $L_{i,j}$ of L_i, if $o_{min}(S_i(n))$, the minimum-scored object in $S_i(n)$, has a score less than or equal to $\mathcal{F}_i(n)$, then $o_{min}(S_i(n))$ is the minimum-scored object in L_i. More precisely, Fagin et al. [11] proved the following corollary.

Corollary 2: [11] Let $o_{min}(S_i(n))$ be the minimum-scored object in $S_i(n)$. Under any monotone linear function f, if $f(o_{min}(S_i(n))) \le \mathcal{F}_i(n)$, then $f(o_{min}(S_i(n))) = f(o_{min}(L_i))$.

Based on Corollary 2, Fagin et al. [11] proposed the TA algorithm. In Figure 4 we show a modified version of the TA algorithm identifying $o_{min}(L_i)$ by retrieving the first few objects from each list $L_{i,j}$. In the algorithm, we assume that the function *getNextObjects*() incrementally retrieves the next object of each list $L_{i,j}$ in L_i starting from the head as has been proposed by Fagin et al. [11]. For example, the first time that *getNextObjects*() is called on L_1 in Figure 2(d), it returns the top objects t_2 and t_4 of the lists $L_{1,1}$ and $L_{1,2}$, respectively. The second time *getNextObjects*() is called on L_1, the next objects, t_1 and t_2, are returned.[4] Starting from $n = 1$, the algorithm incrementally builds $S_i(n)$ by retrieving the next objects in $L_{i,j}$'s in Line 4 until $f(o_{min}(S_i(n)))$ becomes less than or equal to the threshold value $\mathcal{F}_i(n)$. Then in Line 6, the algorithm returns $o_{min}(S_i(n))$ as the $o_{min}(L_i)$. Since the algorithm exits from the while loop only when $f(o_{min}(S_i(n))) \leq \mathcal{F}_i(n)$, from Corollary 2, we can see that the returned object is the minimum-scored object in L_i.

```
1. $n := 0; S_i(n) := \{\}$
2. DO BEGIN
3.     $n := n+1$
4.     $S_i(n) := S_i(n) \cup getNextObjects(L_i, SUB, f)$
5. END WHILE $(f(o_{min}(S_i(n))) > F_i(n))$
6. RETURN $o_{min}(L_i) := o_{min}(S_i(n))$
```

Fig. 4. A modified TA algorithm identifying $o_{min}(L_i)$ from L_i.

2) Identifying $H(i)$: The set $H(i) = \{o | f(o) \leq f(o_{min}(L_i))$ for $o \in L_1 \cup L_2 \cup \cdots \cup L_i\}$ can be obtained similarly, based on the following corollary.

Corollary 3: Let f be an arbitrary monotone linear function, and $f(o_{min}(L_i))$ be the minimum score among all objects in the layer L_i. For each layer L_j $(1 \leq j \leq i)$, let n_j be the minimum n that satisfies $\mathcal{F}_j(n) > f(o_{min}(L_i))$. Then $H(i)$ is a subset of $S_1(n_1) \cup S_2(n_2) \cup \cdots \cup S_i(n_i)$.

Proof: Let o be an object in $H(i)$. From the definition of $H(i)$, $H(i)$ is a subset of $L_1 \cup \cdots \cup L_i$, so o must be in one of $L_1, L_2, \ldots, L_i$. Let o be in L_j $(1 \leq j \leq i)$. From the definition of $H(i)$, o satisfies $f(o) \leq f(o_{min}(L_i))$. From the definition of n_j, Theorem 2, and the condition $\mathcal{F}_j(n) > f(o_{min}(L_i))$, such an o cannot be in $U_j(n_j)$, so it must be in $S_j(n_j)$. Since $S_j(n_j) \subseteq S = S_1(n_1) \cup \cdots \cup S_i(n_i)$, o must be in S. That is, all the objects in $H(i)$ exist in S. Thus, $H(i) \subseteq S$. ■

Corollary 3 suggests the algorithm in Figure 5 that computes the $H(i)$ by retrieving the first few objects in each list $L_{i,j}$ from the layers L_1 through L_i. The algorithm takes $f(o_{min}(L_i))$ (that can be computed by the modified TA algorithm in Figure 4) as its input and returns $H(i)$ as its output. Starting from $j = 1$, it finds the minimum n_j that satisfies $\mathcal{F}_j(n_j) > f(o_{min}(L_i))$ in the while loop between Lines 3 and 6. In the loop it also constructs $S_j(n_j)$ by incrementally retrieving the next objects through *getNextObjects*(). That is, by using $f(o_{min}(L_i))$ as the bound in retrieving more objects from L_j, we can identify the objects in L_j whose scores are lower than or equal to $f(o_{min}(L_i))$ without retrieving all the objects in L_j. Once all such $S_j(n_j)$'s are computed for $1 \leq j \leq i$, it returns $H(i)$ from $S_1(n_1) \cup \cdots \cup S_i(n_i)$ in Line 8. From Corollary 3, it is easy to see that the $H(i)$ returned from the algorithm is correct. Now we are ready to introduce our algorithm that performs both layer-level filtering and list-level filtering using the HL-index.

```
1. FOR $j$ = 1 TO $i$ DO BEGIN
2.     $n_j := 0; S_j(n_j) := \{\}$
3.     DO BEGIN
4.         $n_j := n_j + 1$
5.         $S_j(n_j) := S_j(n_j) \cup getNextObjects(L_j, SUB, f)$
6.     END WHILE $(\mathcal{F}_j(n_j) \leq f(o_{min}(L_i)))$
7. END /* FOR */
8. RETURN $H(i) := \{o \mid f(o) \leq f(o_{min}(L_i))$ for $o \in S_1(n_1) \cup \ldots \cup S_i(n_i)\}$
```

Fig. 5. The algorithm that identifies $H(i)$ from L_1 through L_i.

C. Basic Algorithm

Figure 6 shows *BasicLayerbasedThresholdAlgorithm* (simply, *BasicLTA*) for processing subspace top-k queries using the HL-index. The inputs to BasicLTA are the HL-index and a query $Q = (SUB, f_{\bar{w}}(), k)$. The output is the k objects having the lowest scores for the scoring function $f_{\bar{w}}()$. Starting from $i = 1$, the algorithm first computes $o_{min}(L_i)$ in Lines 2 through 6; it retrieves the next objects from $L_{i,j}$'s by calling *getNextObjects*() and incrementally builds $S_i(n_i)$ until $f_{\bar{w}}(o_{min}(S_i(n_i)))$ becomes lower than or equal to the threshold value $\mathcal{F}_i(n_i)$. (Note the similarity of this part of BasicLTA to the algorithm in Figure 4.) Once the algorithm reaches Line 7, $o_{min}(S_i(n_i))$ is $o_{min}(L_i)$. Then, in Lines 8 through 13, the algorithm computes $H(i)$: for each lower layer L_l $(1 \leq l < i)$, it retrieves next objects from each $L_{l,j}$ and incrementally builds $S_l(n_l)$ until the threshold value $\mathcal{F}_l(n_l)$ becomes greater than $f_{\bar{w}}(o_{min}(S_i(n_i)))$ (which is the same as $f_{\bar{w}}(o_{min}(L_i))$). We note that when the function *getNextObjects()* is called on L_l in Line 11, the function resumes where it left off. It does not start reading the first object from each list again. Then in Lines 15 and 16, the algorithm checks whether or not top-k objects are found. If $S_1(n_1) \cup \cdots \cup S_i(n_i)$ contains k or more objects whose scores are lower than or equal to $f_{\bar{w}}(o_{min}(S_i(n_i)))$ (i.e., if $|H(i)| \geq k$), the algorithm returns the top-k objects in $S_1(n_1) \cup \cdots \cup S_i(n_i)$. Otherwise, it increases i by one and repeats the process.

1) Dealing with Non-Monotone Linear Functions: Before we provide the formal correctness proof of BasicLTA, we first explain the function *getNextObjects*() in more detail.

First, to make our discussion simple, we have assumed that *getNextObjects*() retrieves one object from every list $L_{i,j}$ in L_i. But this is clearly not necessary. Since the zero-weight attributes do not affect the final object score, the function *getNextObjects*() needs to retrieve objects only from the *non-zero-weight attribute* lists.

Second, when the preference vector $\bar{w}$ has negative as well as positive weight terms, the linear function $f_{\bar{w}}$ becomes non-monotone, making it necessary to adjust *getNextObjects*()

[4] What the function *getNextObjects*() does is slightly more complicated than what we describe here because it should handle *non-monotone* linear functions as well. The exact mechanism of *getNextObjects*() will be explained in Section V-C.1.

```
Algorithm BasicLayerbasedThresholdAlgorithm(BasicLTA):
Input: (1) [$L_1 = \{L_{1,1}, \ldots, L_{1,d}\}, \ldots, L_m = \{L_{m,1}, \ldots, L_{m,d}\}$]: an HL Index
       (2) $Q = (SUB, f_{\bar{w}}(), k)$: a subspace top-$k$ query
Output: [top-1, …, top-$k$]: the sequence of $k$ objects with the lowest scores
        for $f_{\bar{w}}()$
Algorithm:
1.  FOR $i$ := 1 to $m$ DO BEGIN /* $m$ is the number of layers */
2.      $n_i$ := 0; $S_i(n_i)$ := {}
3.      DO BEGIN /* Compute $o_{min}(L_i)$ */
4.          $n_i := n_i + 1$
5.          $S_i(n_i) := S_i(n_i) \cup$ getNextObjects($L_i$, $SUB$, $f_{\bar{w}}()$)
6.      END WHILE ($f_{\bar{w}}(o_{min}(S_i(n_i))) > \mathcal{F}_i(n_i)$)
7.      /* Retrieve more objects to compute $H(i)$ */
8.      FOR $l$ = 1 TO $i$ - 1 DO BEGIN /* If $i \geq 2$, perform this loop */
9.          WHILE ($\mathcal{F}_l(n_l) \leq f_{\bar{w}}(o_{min}(S_i(n_i)))$) DO BEGIN
10.             $n_l := n_l + 1$
11.             $S_l(n_l) := S_l(n_l) \cup$ getNextObjects($L_l$, $SUB$, $f_{\bar{w}}()$)
12.         END /* WHILE */
13.     END /* FOR */
14.     /* Does $H(i)$ contains $k$ or more objects ? */
15.     IF ($S_1(n_1) \cup \ldots \cup S_i(n_i)$) contains $k$ or more objects whose scores are
                $f_{\bar{w}}(o_{min}(S_i(n_i)))$ or lower) THEN
16.         RETURN the top-$k$ objects in $S_1(n_1) \cup \ldots \cup S_i(n_i)$
17. END /* FOR */
```

Fig. 6. The BasicLTA algorithm for processing subspace top-k queries using HL-index.

appropriately. We note that a linear function, $f_{\bar{w}}(t)$, consists of d terms, $w[1] * t[1], \ldots, w[d] * t[d]$, as shown in Eq. (1). If $w[j] < 0$, the term $w[j] * t[j]$ decreases as $t[j]$ increases. Otherwise, $w[j] * t[j]$ increases as $t[j]$ does. That is, if we access $L_{i,j}$ from the head (i.e., smallest A_j object first) without considering the sign of $w[j]$'s, we cannot guarantee that the threshold value monotonously increases as we access more objects and, accordingly, cannot exploit Theorem 2. Fortunately, we note that $w[j] * t[j]$ increases as $t[j]$ decreases when $w[j] < 0$. Thus, if we access $L_{i,j}$ from the tail (i.e., largest A_j object first) when $w[j] < 0$, the threshold value monotonously increases. That is, to ensure that $w[j] * t[j]$ monotonously increases as we access more objects, we access $L_{i,j}$ starting from either the head or the tail depending on the sign of $w[j]$ – if the sign is positive, we access objects of $L_{i,j}$ in the ascending order of their A_j attribute values. Otherwise, we access them in the descending order. We call this updated access mechanism *monotone access*.

We use alternative definitions of $S_{i,j}(n)$ and $a_{i,j}(n)$ in Table II depending on the sign of $w[j]$. If $w[j] \geq 0$, we use the "head" versions of the definitions in Table II. Otherwise, we use the "tail" versions. That is, $S_{i,j}(n)$ is the set of the n objects from the head or tail of $L_{i,j}$, and $a_{i,j}(n)$ is the A_j attribute value of the n^{th} object from the head or tail of $L_{i,j}$. Alternatively, we can consider $S_{i,j}(n)$ as the set of the objects that are retrieved by accessing $L_{i,j}$ n times in monotone access. Similarly, we can consider $a_{i,j}(n)$ as the A_j value of the n^{th} object retrieved from $L_{i,j}$ in monotone access. Finally, we set $\mathcal{F}_i(n) = f_{\bar{w}}(a_{i,1}(n), a_{i,2}(n), \ldots a_{i,d}(n))$ and $U_i(n) = L_i - S_i(n)$ under the updated definitions of $a_{i,j}(n)$ and $S_i(n)$. Here, we note that $w[j] * a_{i,j}(n)$ monotonously increases as n increases regardless of the sign of $w[j]$. Using this notation, we naturally have the following corollary:

Corollary 4: Under any *linear* function $f_{\bar{w}}()$, every object in $U_i(n)$ has a score larger than or equal to the threshold value $\mathcal{F}_i(n)$. That is, $\forall o \in U_i(n)$, $f_{\bar{w}}(o) \geq f_{\bar{w}}(a_{i,1}(n), \ldots, a_{i,d}(n)) = \mathcal{F}_i(n)$.

Given Corollary 4, we can see that Corollaries 2 and 3 are still true if we replace the "monotone linear function f" with the "linear function $f_{\bar{w}}$." Let us call them modified Corollary 2 and modified Corollary 3, respectively.

Figure 7 shows the complete version of *getNextObjects*() that performs the monotone access to deal with both monotone and non-monotone linear functions. In Line 3, for each $L_{i,j}$ ($j \in SUB$), it checks the sign of the j^{th} attribute weight. If the sign is positive, it retrieves the next object of $L_{i,j}$ from the head. Otherwise, it retrieves that object from the tail.

```
Function getNextObjects()
Input: (1) $L_i$: the $d$ sorted list $L_{i,1}, \ldots, L_{i,d}$ of $L_i$
       (2) $SUB$: the set of the sequence numbers of the attributes mentioned
           in the query
       (3) $f_{\bar{w}}()$: the linear score function
Output: $S$: the set of the next objects of $L_{i,j}$'s ($j \in SUB$)
1.  $S$ := {}
2.  FOR EACH $j \in SUB$ DO BEGIN
3.      IF $w[j] \geq 0$ THEN /* the sign of $w[j]$ is positive */
4.          Get the next identifier $r$ from $L_{i,j}$ /* starting from the head */
5.      ELSE /* the sign of $w[j]$ is negative */
6.          Get the next identifier $r$ from $L_{i,j}$ in the reverse order /*starting
            from the tail */
7.      Retrieve the object $o$ that has the identifier $r$ from the target relation $R$
8.      $S := S \cup \{o\}$
9.  END /* FOR */
10. RETURN $S$
```

Fig. 7. A function for the BasicLTA algorithm.

We are now ready to prove the correctness of BasicLTA.

Theorem 3: For any linear scoring function $f_{\bar{w}}$, the algorithm *BasicLTA* correctly finds k objects with lowest scores.

Proof: Let R be the set of all objects in the database. Since every returned object has a score $f_{\bar{w}}(o_{min}(S_i(n_i)))$ or less, we can prove correctness by showing that any object o in R but not in $S_1(n_1) \cup \cdots \cup S_i(n_i)$ at Line 16 satisfies $f_{\bar{w}}(o) \geq f_{\bar{w}}(o_{min}(S_i(n_i)))$. We first note that, due to the condition in Line 6, $f_{\bar{w}}(o_{min}(S_i(n_i))) \leq \mathcal{F}_i(n_i)$ when the algorithm reaches Line 16. Given this inequality, we know from modified Corollary 2 that

$$f_{\bar{w}}(o_{min}(S_i(n_i))) = f_{\bar{w}}(o_{min}(L_i)). \quad (2)$$

Consider three cases of o in R but not in $S_1(n_1) \cup \cdots \cup S_i(n_i)$:

(1) $o \in L_i$: From Eq. (2), $o_{min}(S_i(n_i))$ is the minimum-scored object in L_i, so $f_{\bar{w}}(o) \geq f_{\bar{w}}(o_{min}(S_i(n_i)))$.

(2) $o \in L_l$ for $l > i$: From Theorem 1, we know that $f_{\bar{w}}(o) \geq f_{\bar{w}}(o_{min}(L_i))$, so from Eq. (2), $f_{\bar{w}}(o) \geq f_{\bar{w}}(o_{min}(S_i(n_i)))$.

(3) $o \in L_l$ for $l < i$: Due to the condition in Line 9, n_l is increased until

$$\mathcal{F}_l(n_l) > f_{\bar{w}}(o_{min}(S_i(n_i))). \quad (3)$$

Because $o \notin S_l(n_l)$, o should be in $U_l(n_l)$ by the definition of $U_l(n_l) = L_l - S_l(n_l)$. From Corollary 4, such an o satisfies

$$f_{\bar{w}}(o) \geq \mathcal{F}_l(n_l). \quad (4)$$

From Eqs. (3) and (4), we get $f_{\bar{w}}(o) > f_{\bar{w}}(o_{min}(S_i(n_i)))$. ■

D. Enhanced Algorithm

In this section, we enhance the BasicLTA algorithm to further reduce the number of retrieved objects. In BasicLTA, once we identify $o_{min}(L_i)$ from layer L_i, we retrieve more objects in layers L_1 through L_{i-1} using $f_{\bar{w}}(o_{min}(L_i))$ as the bound of their scores. Our following observation suggests that we may be able to use a smaller number as this bound and retrieve fewer objects from L_1 through L_i:

Lemma 1: During the execution of BasicLTA, the inequality $\min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i)))) \leq f_{\bar{w}}(o_{min}(L_i))$ always holds.

Proof: $o_{min}(L_i)$ is in $S_i(n_i)$ or $U_i(n_i)$. If $o_{min}(L_i) \in S_i(n_i)$, then $f_{\bar{w}}(o_{min}(S_i(n_i))) = f_{\bar{w}}(o_{min}(L_i))$. If $o_{min}(L_i) \in U_i(n_i)$, then $\mathcal{F}_i(n_i) \leq f_{\bar{w}}(o_{min}(L_i))$ from Corollary 4. Thus, $\min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i)))) \leq f_{\bar{w}}(o_{min}(L_i))$. ■

From Lemma 1 and Theorem 1, it is easy to see that every object in layers L_{i+1} through L_m has a score $\min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i))))$ or higher. Therefore, if we can find k or more objects from layers L_1 through L_i with score $\min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i))))$ or less, they are guaranteed to be the top-k, because no object in L_{i+1} and above has a smaller score. Since we can use the *minimum* between $\mathcal{F}_i(n_i)$ and $f_{\bar{w}}(o_{min}(S_i(n_i)))$ as the bound without having to identify $o_{min}(L_i)$, we retrieve fewer objects from L_1 through L_i to compute the top-k objects.

Figure 8 shows *EnhancedLayerbasedThresholdAlgorithm* (simply, *EnhancedLTA*), which implements this idea. The inputs and the output of EnhancedLTA are same to those of BasicLTA in Figure 6. In BasicLTA, starting from $i = 1$, we keep retrieving objects from L_i until we find the $o_{min}(L_i)$ by repeatedly calling *getNextObjects*() until $\mathcal{F}_i(n_i) > f_{\bar{w}}(o_{min}(S_i(n_i)))$. Only then it goes back to previous layers to retrieve more objects with the score bound $f_{\bar{w}}(o_{min}(L_i))$. In contrast, in EnhancedLTA, each time we call *getNextObjects*() in Line 5, we immediately go back to the earlier layers and retrieve more objects with the score bound $\min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i))))$ (Lines 7 through 12). If there exist k or more such objects, we return the top-k objects in Line 16. We now prove the correctness of EnhancedLTA.

Theorem 4: For any linear scoring function $f_{\bar{w}}$, the algorithm $EnhancedLTA$ correctly finds k objects with lowest scores.

Proof: This proof is very similar to our proof for Theorem 3. Let R be the set of all objects in the database. Since every returned object has a score $\min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i))))$ or less, we can prove the correctness by showing that any object o in R but not in $S_1(n_1) \cup \cdots \cup S_i(n_i)$ at Line 15 satisfies $f_{\bar{w}}(o) \geq \min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i))))$.

```
Algorithm EnhancedLayerbasedThresholdAlgorithm(EnhancedLTA):
Input: (1) [$L_1 = \{L_{1,1}, \ldots, L_{1,d}\}, \ldots, L_m = \{L_{m,1}, \ldots, L_{m,d}\}$]: an HL Index
       (2) $Q = (SUB, f_{\bar{w}}(), k)$: a subspace top-$k$ query
Output: [top-1, ..., top-$k$]: the sequence of $k$ objects with the lowest scores
        for $f_{\bar{w}}()$
Algorithm:
1.  FOR $i$ := 1 to $m$ DO BEGIN /* $m$ is the number of layers */
2.      $n_i := 0$; $S_i(n_i) := \{\}$
3.      DO BEGIN
4.          $n_i := n_i + 1$
5.          $S_i(n_i) := S_i(n_i) \cup getNextObjects(L_i, SUB, f_{\bar{w}}())$
6.          /* Retrieve more objects */
7.          FOR $l$ = 1 TO $i$ - 1 DO BEGIN /* If $i \geq 2$, perform this loop */
8.              WHILE ($\mathcal{F}_l(n_l) \leq \min(f_{\bar{w}}(o_{min}(S_i(n_i))), \mathcal{F}_i(n_i))$) DO BEGIN
9.                  $n_l := n_l + 1$
10.                 $S_l(n_l) := S_l(n_l) \cup getNextObjects(L_l, SUB, f_{\bar{w}}())$
11.             END /* WHILE */
12.         END /* FOR */
13.         /* Does $S_1(n_1) \cup \ldots \cup S_i(n_i)$ contains $k$ or more objects ? */
14.         IF ($S_1(n_1) \cup \ldots \cup S_i(n_i)$ contains $k$ or more objects whose scores
                    are $\min(f_{\bar{w}}(o_{min}(S_i(n_i))), \mathcal{F}_i(n_i))$ or lower) THEN
15.             RETURN the top-$k$ objects in $S_1(n_1) \cup \ldots \cup S_i(n_i)$
16.     END WHILE ($f_{\bar{w}}(o_{min}(S_i(n_i))) > \mathcal{F}_i(n_i)$)
17. END /* FOR */
```

Fig. 8. The query processing algorithm enhanced to use a tighter bound.

Consider three cases of o in R but not in $S_1(n_1) \cup \cdots \cup S_i(n_i)$:
(1) $o \in L_i$: Since $o_{min}(L_i)$ is the minimum-scored object in L_i, $f_{\bar{w}}(o) \geq f_{\bar{w}}(o_{min}(L_i)) \geq \min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i))))$ from Lemma 1.
(2) $o \in L_l$ for $l > i$: From Theorem 1, we know that $f_{\bar{w}}(o) \geq f_{\bar{w}}(o_{min}(L_i)) \geq \min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i))))$.
(3) $o \in L_l$ for $l < i$: Due to the condition in Line 8, n_l is increased until

$$\mathcal{F}_l(n_l) > \min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i)))). \quad (5)$$

Because $o \notin S_l(n_l)$, o should be in $U_l(n_l)$ by the definition of $U_l(n_l) = L_l - S_l(n_l)$. From Corollary 4, such an o satisfies $f_{\bar{w}}(o) \geq \mathcal{F}_l(n_l) > \min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i))))$ from Eq. (5). ■

E. HL-index for Monotone Functions

We now briefly explain how we may extend the HL-index to handle all monotone functions including both linear and non-linear cases. In our earlier algorithms, the core part that makes the linearity assumption is the layering step of the index construction algorithm. With a non-linear scoring function, top-k objects may not necessarily reside in the first k layers built through the recursive extraction of convex hull vertices, making it impossible to apply layer-level filtering. To address this problem for monotone non-linear functions, we build layers by recursively drawing *skylines* [5] instead of convex hulls. Since the layer list consisting of skylines satisfies the optimally linearly ordered set property (extended for monotone functions) as proved in Lemma 2 below, the same theorems and lemmas that we proved in Section V also apply to the skyline version with the correctness of the algorithms proved.

Therefore, this new version of the HL-index can use the identical algorithm in Fig. 8 except that the layers are now constructed using skylines, not convex hull vertices.

Lemma 2: For a given set U of objects in the universe, a layer list constructed using skylines over U satisfies optimally linearly ordered set property under any monotone function f.

Proof: Let $L = \{L_1, L_2, \ldots, L_m\}$ be the list of layers constructed by recursive extraction of skylines. That is, $L_1 \cup \cdots \cup L_m$ is the entire database, and L_i is the set of objects in the skyline over $L_i \cup \cdots \cup L_m$. According to the definition of the skyline [5] [5], when L_i is the skyline over $L_i \cup \cdots \cup L_m$, every object in $L_{i+1} \cup \cdots \cup L_m$ is dominated by at least one object in L_i. That is, for any $o \in L_{i+1} \cup \cdots \cup L_m$, there exist $o' \in L_i$ such that o is dominated by o'. Then $f(o') \leq f(o)$ for any monotone function f according to the definition of monotony. That is, $\forall o \in L_{i+1} \cup \cdots \cup L_m, \quad \exists o' \in L_i$ s.t. $f(o') \leq f(o)$. Therefore, $f(o_{min}(L_i)) \leq f(o)$ for any $o \in L_j$ $(i < j \leq m)$. This proves the optimally linearly ordered set property of L. ■

From now on, if we need to differentiate this new version of the HL-index from the earlier one, we refer to the earlier one as HL-index (convex) and this new one as HL-index (skyline).

VI. Experiments

A. Experimental Data and Environment

We compare the index building time and the query performance of the HL-index with the following existing methods: ONION [7] (a layer-based method), TA [11] (a list-based method), PREFER [14] (a view-based method) and SUB-TOPK [19]. We use the wall clock time as the measure of the index building time and the number of objects read from database, *Num_Objects_Read*, as the measure of the query performance. *Num_Objects_Read* is a measure widely used in top-k research [7], [14], [21] because its results are not affected by the implementation detail of the individual methods used in the experiments. 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 datasets. For the synthetic dataset, we generate uniform datasets (*UNIFORM*) by using the generator used by Borzsonyi et al. [5]. The datasets consist of three-, five-, and seven-dimensional datasets of 10K, 100K, and 1000K objects. For the real dataset, we use the regular season statistics [6] of the NBA players that play over ten minutes per year from 1951 to 2007 (*NBA*). The dataset consists of 19364 objects with seven attributes: minutes played, total points, field goals attempted, free throws attempted, total rebounds, total assists, and total personal fouls.

[5] "The *Skyline* is defined as the set of 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." [5]

[6] http://www.basketballreference.com

For the experiments, we have implemented HL-index (both convex hull and skyline based versions), ONION, TA, SUB-TOPK, and PREFER using C++. For TA, we use the TA algorithm extended with our monotone access method (to handle non-monotone linear functions). We call it *TA(e)*. For PREFER, we translate the code of the PREFER system [7] witten in JAVA into C++. To compute convex hulls for HL-index and ONION, we used the Qhull library [1]. To compute skylines, we used the BNL algorithm [5]. We conducted all the experiments on a Pentium-4 2.0 GHz Linux PC with 1GBytes of main memory.

B. Index building time

We first compare the index building times of HL-index, ONION, and TA(e). Due to space limit, we only report the results for the UNIFORM data of the dimension $d = 5$ while varying the number of objects (N = 10K, 100K, 1000K) in Table III. From these results, we observe that the index building times of ONION and TA(e) are (almost) equivalent to the times of the layering step and the listing step, respectively. We also observe that for all ranges of N, the layering step takes significantly longer than the listing step, and thus, the total index building time of HL-index is very close to the building time of ONION. The results for other parameter settings is close to what we observe from Table III. Similarly, the index building time of HL-index(skyline) is close to the sum of the time for constructing the skyline layers and the time for building the lists of TA(e).

TABLE III
Index building time as N is varied (UNIFORM and d=5).

Methods (sec)		10K	100K	1000K
ONION		9.14	152.70	2500.61
TA		0.24	2.89	35.24
HL-index	layering	9.14	152.70	2500.61
	listing	0.23	2.65	31.57
HL-index (skyline)	layering	1.33	146.99	17283.95
	listing	0.24	2.46	39.36

C. Performance of monotone or non-monotone linear queries

We now compare the query performance of the HL-index against other existing methods under both monotone and non-monotone *linear* functions. Note that while ONION and TA(e) support all *linear* functions, PREFER and SUB-TOPK support *monotone* functions and thus cannot handle non-monotone linear functions. In this section, therefore, we only compare HL-index (that uses the convex hall layering), ONION, and TA(e). The comparison of all six methods will be done in the next section when we use *monotone linear* functions.

We measure the query performance of the three methods on the synthetic and real datasets while varying the sub-dimension s (i.e., the size of *SUB* in Section III), the number of results k, N, and d. We measure $Num\_Objects\_Read$ for ten randomly generated queries having different preference vectors, and then, use the average value over all queries. We first generate the set *SUB*, which is the subset of the sequence numbers

[7] http://db.ucsd.edu/PREFER/application.htm

of the attributes, for each size $s\,(1 \le s \le d)$. That is, we randomly choose s unique elements from $\{1, 2, \ldots, d\}$. Then, we randomly choose the attribute preference $w[i]$, which is the weight of the i^{th} attribute $(i \in \mathit{SUB})$ in the preference vector $\bar{w}$, from $\{-4, -3, -2, -1, 1, 2, 3, 4\}$, and normalize $w[i]$'s so that $\sum_{i \in \mathit{SUB}} |w[i]| = 1$. Table IV summarizes the experiments and the parameters used for this set of experiments.

TABLE IV
EXPERIMENTS AND PARAMETERS USED FOR COMPARING THE PERFORMANCE OF MONOTONE OR NON-MONOTONE LINEAR QUERIES.

Experiments		Parameters	
Exp. 1	comparison of the query performance as N is varied	dataset	UNIFORM
		N	10K, 100K, 1000K
		d	5
		s	3
		k	50
Exp. 2	comparison of the query performance as s is varied	dataset	UNIFORM
		N	100K
		d	5
		s	1, 2, …, 5
		k	50
Exp. 3	comparison of the query performance as k is varied	dataset	UNIFORM
		N	100K
		d	5
		s	3
		k	1, 10, 20, …, 100
Exp. 4	comparison of the query performance as d is varied	dataset	UNIFORM
		N	10K, 100K, 1000K
		d	3, 5, 7
		s	$\lceil \frac{d}{2} \rceil$
		k	50
Exp. 5	comparison of the query performance as s is varied using a real dataset	dataset	NBA (real data)
		N	19364
		d	7
		s	1, 2, …, 7
		k	50
Exp. 6	comparison of the query performance as k is varied using a real dataset	dataset	NBA (real data)
		N	19364
		d	7
		s	4
		k	1, 10, 20, …, 100

Comparison of basic and enhanced algorithm
Before we compare the HL-index with other approaches, we first show the improvement of EnhandedLTA compared to BasicLTA. Figure 9 shows the query performance of HL-index and HL-index(basic) as s is varied from 1 to 3, where HL-index(basic) and HL-index represent the results from BasicLTA and EnhancedLTA, respectively. HL-index improves performance by up to 20% over HL-index(basic). Due to this better performance, we show only HL-index from EnhancedLTA in the rest of our experimental results.

Experiment 1: query performance as N is varied
Figure 10 shows the query performance of HL-index, ONION, and TA(e) as N is varied from 10K to 1000K. From the result, we observe that HL-index outperforms ONION and TA(e) for all N values. For example, HL-index outperforms both ONION and TA(e) by a factor of three or more at N=1000K. Interestingly, we note that TA(e) performs quite well for a small N (it shows performance close to HL-index for N=10K), but its performance gets significantly worse than the two others beyond a certain cross-over point. This is because the number of objects retrieved from each list of TA(e) grows linearly with N, while the number of objects in each layer of ONION increases sublinearly (more precisely, in proportion to $(\ln N)^{c1}$ where $c1$ is a positive constant [3]). Since the HL-index exploits the synergic effect of the layer-level filtering and the list-level filtering due to its meticulous integration of the two filtering capabilities, the HL-index significantly outperforms both TA(e) and ONION for large N values, making it particularly useful for a large database.

Experiment 2: query performance as s is varied
Figure 11 shows the query performance of HL-index, ONION, and TA(e) as s is varied from 1 to 5. When s = 1, the query performance of HL-index is worse than that of TA(e), but is much better than that of ONION. As mentioned in Section IV, in HL-index, the elements within a layer are totally ordered, but elements in different layers are not. Thus, HL-index reads more than k objects from the heads (or tails) of lists in some layers while TA(e) only reads k objects from the head (or tail) of one sorted list because the elements of the list are totally ordered. However, when $s \ge 2$, HL-index begins to show better performance than the other methods due to the synergic effect of the two filtering capabilities. HL-index improves by 1.4 to 166.2 times over ONION and by 0.5 to 2.6 times over TA(e).

Experiment 3: query performance as k is varied
Figure 12 shows the query performance of HL-index, ONION, and TA(e) as k is varied from 1 to 100. HL-index outperforms the other methods for all k values. HL-index improves by 2.7 to 3.2 times over ONION and by 1.6 to 5.0 times over TA(e).

Experiment 4: query performance as d is varied
Figure 13 shows the query performance of HL-index, ONION, and TA(e) as d is varied from 3 to 7. Here, we use $s = 2$ when $d = 3$, $s = 3$ when $d = 5$, and $s = 4$ when $d = 7$. Since $(\ln N)^{d-1}$ is the average number of objects in convex hull vertices [3], the entire objects can reside in the first layer when d is very large. Thus, in that case, since HL-index can exploit only the list-level filtering like TA(e), the performance of HL-index would converge to that of TA(e). However, within the experimental range of d, HL index outperforms TA(e). HL-index improves by 2.5 to 3.2 times over ONION and by 1.2 to 3.0 times over TA(e).

Experiment 5: query performance as s is varied when using a real dataset
Figure 14 shows the query performance of HL-index, ONION, and TA(e) as s is varied from 1 to 7 when using NBA, a seven-dimensional real dataset. HL-index begins to outperform the other methods when $s \ge 2$. This tendency is similar to that of the synthetic dataset shown in Figure 11. HL-index improves by 1.4 to 138.9 times over ONION and by 0.6 to 1.6 times over TA(e).

Experiment 6: query performance as k is varied when using a real dataset
Figure 15 shows the query performance of HL-index, ONION, and TA(e) as k is varied from 1 to 100 when using NBA. HL-index outperforms the other methods for the entire range of k. This tendency is similar to that of the synthetic dataset shown in Figure 12. HL-index improves by 2.3 to 3.3 times over ONION and by 1.3 to 5.3 times over TA(e).

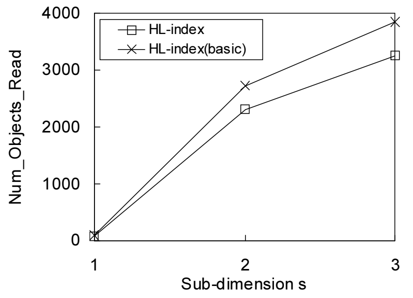


Fig. 9. Query performance of the basic and enhanced algorithms (NBA, d=7, N=19364, and k=50).

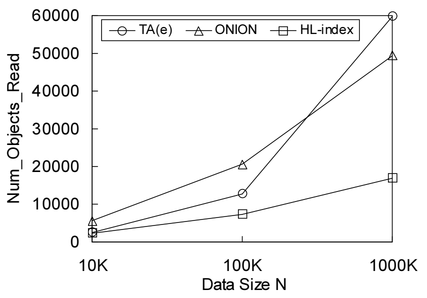


Fig. 10. Query performance as N is varied (UNIFORM, d=5, s=3, and k=50).

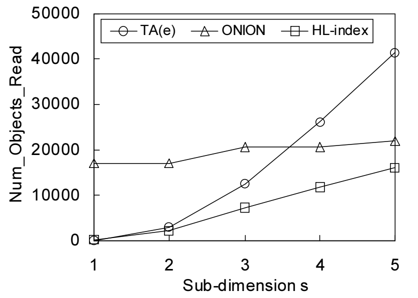


Fig. 11. Query performance as s is varied (UNIFORM, d=5, N=100K, and k=50).

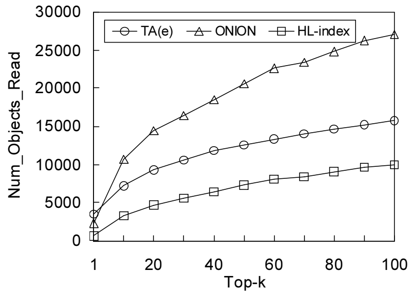


Fig. 12. Query performance as k is varied (UNIFORM, d=5, s=3, and N=100K).

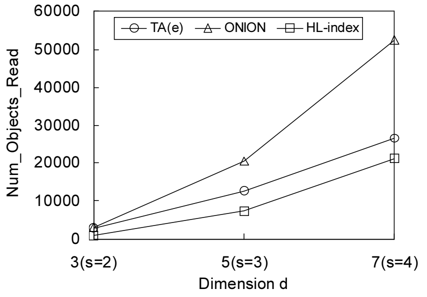


Fig. 13. Query performance as d is varied (UNIFORM, N=100K, and k=50).

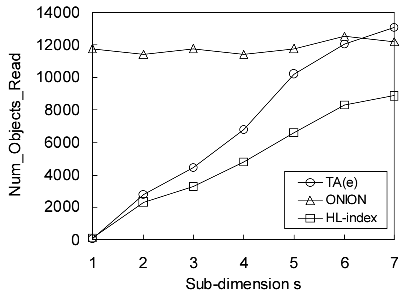


Fig. 14. Query performance as s is varied (NBA, d=7, N=19364, and k=50).

D. Performance of monotone linear queries

We now compare the performance of all six methods, HL-index (convex), HL-index (skyline), ONION, TA(e), PREFER, and SUB-TOPK, under *monotone linear functions*. We limit the scoring function to monotone linear functions because HL-index (skyline), PREFER, and SUB-TOPK are designed to handle monotone functions and cannot deal with non-monotone linear functions.

In generating the queries, we use the same setting that we used in the previous section, except that we now choose the attribute preference $w[i]$ randomly from $\{1, 2, 3, 4\}$, and normalize them to be $\sum_{i \in SUB} |w[i]| = 1$. We note that, in the previous section, $w[i]$ were chosen from $\{-4, -3, -2, -1, 1, 2, 3, 4\}$ to allow non-monotone linear functions. Again, we measure $Num\_Objects\_Read$ for ten randomly generated queries, and then, use the average value over them. Table V summarizes the experiments and the parameters used for this set of experiments.

TABLE V
EXPERIMENTS AND PARAMETERS USED FOR COMPARING THE PERFORMANCE OF MONOTONE LINER QUERIES.

Experiments		Parameters	
Exp. 7	comparison of the query performance as s is varied when using only monotone linear functions	dataset	UNIFORM
		N	100K
		d	5
		s	1, 2, …, 5
		k	50
Exp. 8	comparison of the query performance as k is varied when using only monotone linear functions	dataset	UNIFORM
		N	100K
		d	5
		s	3
		k	1, 10, 20, …, 100

Experiment 7: query performance as s is varied when using only monotone linear scoring functions
Figure 16 shows the query performance of HL-index (convex), HL-index (skyline), ONION, TA(e), PREFER, and SUB-TOPK as s is varied from 1 to 5. Since the query performance of PREFER improves as the number of views increases, for a fair comparison with HL-index, we use five views generated randomly. [8] The comparison between HL-index, ONION, and TA(e) shows similar results to what we observed earlier; HL-index shows significant improvement over ONION and TA(e) in almost all cases. For example, HL-index (skyline) also improves by up to 782.0 times over PREFER and by up to 2.6 times over SUB-TOPK. In addition, for monotone *non-linear* queries [9], HL-index (skyline) also shows similar results to what we observed in this experiment. HL-index (skyline) improves by up to 1063.8 times over PREFER, by up to 2.5 times over SUB-TOPK, and by up to 3.7 times over TA(e)(the result graph is omitted due to space limit).

Experiment 8: query performance as k is varied when using only monotone linear scoring functions
Figure 17 shows the query performance of HL-index (convex), HL-index (skyline), ONION, TA(e), PREFER, and SUB-TOPK as k is varied from 1 to 100 when using monotone linear scoring functions. Here again, PREFER uses five views. The performance trends of HL-index, ONION, and TA(e) are similar to what we observed in earlier experiments; HL-index shows significant improvement over ONION and TA(e).

[8]We simply consider one view as one list. Thus, PREFER with five views has five lists and HL-index has five lists when $d = 5$.

[9]We use a quadratic function, $f(t) = \sum_{i=1}^{d} w[i] * t[i]^2$, as the monotone non-linear scoring function.

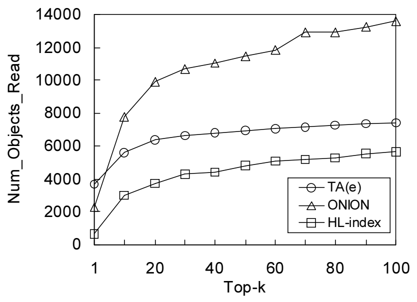


Fig. 15. Query performance as k is varied (NBA, d=7, s=4, N=19364, and k=50).

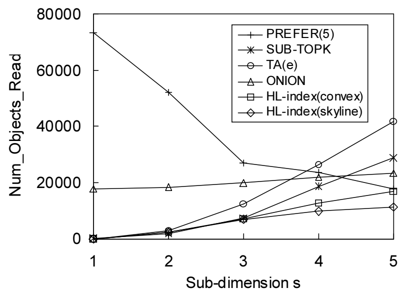


Fig. 16. Performance of monotone linear queries as s is varied (UNIFORM, d=5, N=100K, and k=50).

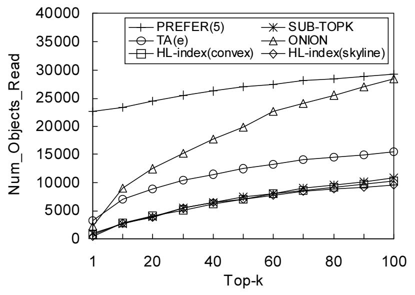


Fig. 17. Performance of monotone linear queries as k is varied (UNIFORM, d=5, s=3, and N=100K).

Compared to SUB-TOPK, HL-index shows approximately 10% (which should be higher with a higher sub-dimension such as $s \geq 4$) performance improvement in many cases. Compared to PREFER, HL-index shows up to 33.6 times performance improvement. For monotone *non-linear* queries, HL-index (skyline) shows similar results to what we observed in this experiment. HL-index (skyline) improves by up to 232.6 times over PREFER and by up to 9.5 times over TA(e) and shows performance comparable to SUB-TOPK (the result graph is omitted due to space limit).

VII. CONCLUSIONS

In this paper, we proposed the HL-index that is designed to handle top-k queries on an arbitrary subset of attributes efficiently. The HL-index has significantly more pruning power than any existing method because it exploits the synergic effect of the integration of layer-level filtering and list-level filtering. We described top-k answer computation algorithms for the HL-index and formally proved their correctness. We also derived a *tight* bound for guaranteeing the correct query results through in-depth analysis.

For the clarity of our exposition, we first presented the convex hull version of the HL-index that deals with linear scoring functions. Since this version of HL-index does not put any restriction on the sign of the attribute weight $w[i]$, it can handle *both monotone and non-monotone linear functions*. We then briefly discussed the skyline version of the HL-index that can handle monotone (linear or non-linear) functions.

Our extensive experiments demonstrate that the HL-index outperforms all existing methods in practically all scenarios, and its improvement gets more noticeable for larger databases. Given that our HL-index retrieves significantly fewer objects than any existing methods, we expect that it is particularly suitable for the environments like main memory DBMSs or the ones using flash memory, which are being and will become more and more prevalent in the foreseeable future [16].

Finally, we note that the HL-index algorithms and their proofs can be applied to any layering methods that satisfy the optimally linearly ordered set property. Finding a layering method that can handle both all linear functions and all monotone functions while satisfying this property would be an interesting problem. We leave this as future study.

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.

REFERENCES

[1] Barber, B., Dobkin, D., and Huhdanpaa, H., "The Quickhull Algorithm for Convex Hulls," *ACM TOMS*, Vol. 22, No. 4, 1996.
[2] Bast, M. et al., "IO-Top-k: Index-access Optimized Top-k Query Processing," In *VLDB*, 2006.
[3] Bentley, J. L. et al., "On the Average Number of Maxima in a Set of Vectors and Applications," *JACM*, Vol. 33, No. 2, 1978.
[4] Berg, M. et al., *Computational Geometry: Algorithms and Applications*, 2nd ed., Springer-Verlag, 2000.
[5] Borzsonyi, S., Kossmann, D., and Stocker, K., "The Skyline Operator," In *ICDE*, 2001.
[6] Chan, C.-Y., Eng, P.-K., and Tan, K.-L., "Stratified computation of skylines with partially-ordered domains," In *SIGMOD*, 2005.
[7] Chang, Y. C. et al., "The Onion Technique: Indexing for Linear Optimization Queries," In *SIGMOD*, 2000.
[8] Chaudhuri, S., Ramakrishnan, R., and Weikum, G.,"Integrating DB and IR Technologies: What is the Sound of One Hand Clapping?," In *CIDR*, 2005.
[9] Das, G. et al., "Answering Top-k Queries Using Views," In *VLDB*, 2006.
[10] Fagin, R., "Fuzzy Queries in Multimedia Database Systems," In *PODS*, 1998.
[11] Fagin, R., Lotem, A., and Naor, M., "Optimal Aggregation Algorithms for Middleware," In *PODS*, 2001.
[12] Gass, S. G., *Linear Programming: Method and Applications*, 5th ed. An International Thomson Publishing Company, 1985.
[13] Hadley, G., *Linear Programming*, Addison-Wesley Publishing Company, 1962.
[14] Hristidis, V. and Papakonstantinou, Y. "Algorithms and applications for answering ranked queries using ranked views," *The VLDB Journal*, Vol. 13, No. 1, 2004.
[15] Heo, J.-S. et al., "The Partitioned-Layer Index: Answering Monotone Top-k Queries Using the Convex Skyline and Partitioning-Merging Technique," *Information Sciences*, to appear, 2009.
[16] Lee, S.-W. et al., "A Case for Flash Memory SSD in Enterprise Database Applications," In *SIGMOD*, 2008.
[17] Papadias, D. et al., "Progressive skyline computation in database systems," *ACM TODS*, Vol. 30, No. 1, 2005.
[18] Tan, K.-L., Eng, P.-K., and Ooi, B. C., "Efficient progressive skyline computation," In *VLDB*, 2001.
[19] Tao, Y., Xiao, X., and Pei, J., "Efficient Skyline and Top-k Retrieval in Subspaces," *IEEE TKDE*, Vol. 19, No. 8, 2007.
[20] Yi, K. et al., "Efficient Maintenance of Materialized Top-k Views," In *ICDE*, 2003.
[21] Xin, D., Chen, C., and Han, J., "Towards Robust Indexing for Ranked Queries," In *VLDB*, 2006.