Data & Knowledge Engineering 83 (2013) 1–19

Contents lists available at SciVerse ScienceDirect

Data & Knowledge Engineering

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

Subspace top-k query processing using the hybrid-layer index with a tight bound

Jun-Seok Heo [a], Junghoo Cho [b], Kyu-Young Whang [a,*]

[a] Korea Advanced Institute of Science and Technology (KAIST), Republic of Korea
[b] University of California, LA, USA

ARTICLE INFO

Article history:
Received 28 April 2011
Received in revised form 10 July 2012
Accepted 14 July 2012
Available online 11 September 2012

Keywords:
Access methods
Query
Top-k queries
Subspaces
Linear scoring functions
Layering
Listing
Hybrid

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. Our experiments demonstrate that the query performance of the HL Index significantly outperforms other state-of-the-art indexes in most scenarios.

© 2012 Published by Elsevier B.V.

1. 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 [1]. 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.
```

* Corresponding author.
E-mail addresses: jsheo@mozart.kaist.ac.kr (J.-S. Heo), cho@cs.ucla.edu (J. Cho), kywhang@mozart.kaist.ac.kr (K.-Y. Whang).

0169-023X/$ – see front matter © 2012 Published by Elsevier B.V.
http://dx.doi.org/10.1016/j.datak.2012.07.001

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. [2,3] 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. [4] 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 list-based 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 *subsets* of attributes and use the appropriate index for a query as in [5,6]. We refer to these approaches as the *view-based approach*. Clearly, view-based approaches lead to high query performance if the "closest" answers to the query issued by a user has been precomputed. Otherwise, they lead to low query performance. They can improve query performance by increasing the number of indexes, but the space overhead increases in proportion to the number of indexes [7].

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 2 and we formally define the top-k queries that we handle in Section 3. Then, in Section 4, we describe the HL-index construction algorithm and, in Section 5, explain the top-k query processing algorithm using the HL-index and prove its correctness. In Section 6 we present our experiments that compare the performance of the HL-index to existing approaches. We conclude the paper in Section 7.

2. 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.

2.1. 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 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. ONION [4], PL-index [8], and AppRI [9] are well-known methods of this approach.

ONION [4] builds the index by making layers with the vertices (or the *extreme points* [10]) of the *convex hulls* [11] 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. [4] 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 [9], particularly when the number of attributes mentioned in the query is small, because it reads all the tuples in a layer. In order to avoid the pitfall, PL-index [8] and AppRI [9] construct a database in more layers than ONION does by taking advantage of the assumption that queries have monotone linear functions. PL-index reduces the layer size by partitioning the single layer list into multiple layer lists and by using the *convex skyline*, which is a combination of the convex hull and the skyline, as the layering scheme. AppRI constructs a list of layers by exploiting the domination relation of skylines. However, to support an arbitrary linear function, PL-index and AppRI have to build 2^d indexes due to the monotone assumption. Here, d represents the number of attributes. In contrast, ONION builds only one index.

2.2. 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 [13,3]. For example, the threshold algorithm (TA) [3,14], 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. In recent, a number of extensions of TA has been proposed for processing top-k queries in a distributed environment (or in a disk-based environment), where sequential/random access cost difference is not negligible [15,16,19].

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.

2.3. View-based approach

The basic idea behind the view-based approach is to “precompute” the answers to a class of queries on subsets 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 [6] and LPTA [5] are well-known methods of this approach. Since the view-based approach requires constructing a number of indexes in order to improve query performance [6], its space and maintenance overhead often becomes an important issue in using this approach for a practical system [7].

2.4. Other approaches

There exists a large body of work for efficient computation of skyline queries [17,18,20–22]. Because the skyline contains at least one tuple that minimizes any monotone scoring function [23], this work can be used to deal with top-k queries under a monotone scoring function for the special case of $k=1$. SUB-TOPK [21] is one extension of these methods that finds the top-k results, 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. In addition, there also exists a body of work for extending the traditional top-k queries. First, to handle uncertain data such as sensing data of sensors [24,25], this work models uncertainty of these data as a probability distribution [24,26] and proposes methods that find k tuples with the highest probability to be the top-k results efficiently [25,27]. Second, to handle spatial objects having both text and location information, this work focuses on constructing indexes to efficiently combine the text relevancy and the location proximity [28]. Third, to handle objects in a number of sources (or subsystems) [29,30], this work identifies the costs of accessing each source and finds the cheapest plan for accessing the sources where the top-k results can be. Last, this work handles *reverse* top-k queries that efficiently find preferences making a given tuple to be ranked high instead of finding top k tuples [31,32].

Table 1 summarizes the top-k indexing methods that are compared in this paper and the functions they support.

Table 1
Top-k indexing methods compared and the functions they support.

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

3. 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 $f(t): t \rightarrow [-1.0,1.0]$ 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 $f(t^1) \leq f(t^2) \leq \ldots \leq f(t^k) \leq f(t^l)$, $k+1 \leq l \leq N$. Here, t^j denotes the jth ranked object in the ascending order of their score, where $1 \leq j \leq N$.

The scoring function for top-k queries is generally assumed to be either linear [4–6,8,9,30] or monotone [5,3,6,8,9,21,29,30,33]. 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 ith attribute value of t and $w[i]$ is the "weight" of the ith 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 [3]: if $t[i] \leq t'[i]$ for all $i = 1, \ldots d$, then $f(t) \leq f(t')$. 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. For clarity 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 5.5.

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 [21]. To emphasize this fact, we use SUB to denote the set of attributes with $w[i] \neq 0$ and call the size of SUB the *sub-dimension* and the space consisting of these attributes the *subspace*. That is, $SUB = \{i \mid w[i] \neq 0 \text{ for } i = 1, \ldots, d\}$. 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 $(SUB, f_{\bar{w}}(), k)$. In Table 2, we summarize the notation that we use throughout this paper. The symbols that have not been introduced yet will be explained in Section 5.

4. 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 ith layer. Every object belongs to one

Table 2
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 $(1 \leq i \leq d)$
t	An object in R (t is considered as a d-dimensional vector $\bar{t}$ that has $t[i]$ as the ith 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 ith element)
$w[i]$	The weight of attribute A_i in the preference vector $\bar{w}$
L_i	The ith layer or the set of objects in the ith 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
$H(i)$	$\{o \mid f_{\bar{w}}(o) \leq f_{\bar{w}}(o_{min}(L_i))$ for $o \in L_1 \cup L_2 \cup \ldots \cup L_i\}$
$S_{i,j}(n)$	The set of the first n objects from the head (or tail) of $L_{i,j}$
$S_i(n)$	$S_{i,1}(n) \cup S_{i,2}(n) \cup \ldots \cup S_{i,d}(n)$
$U_i(n)$	$L_i - S_i(n)$; the objects in L_i that are not in $S_i(n)$
$a_{i,j}(n)$	The A_j value of the nth 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 $\mathcal{F}_i(n)$.

```
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 ≠ {}) 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, ..., 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, ..., L_1,d}, ..., L_m = {L_m,1, ..., L_m,d}]
```

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

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 jth attribute values.

As we mentioned earlier, our HL-index can be built for either all linear functions or for all monotone functions. Fig. 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 ith layer L_i contains d sorted lists, $L_i=\{L_{i,1},\ldots, L_{i,d}\}$. We explain the algorithm using Example 1.

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 Fig. 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 ith layer L_i) as shown in the left-most rectangle in Fig. 2(c). The reason why we use the convex hull to partition R will be explained later in Section 5. 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 Fig. 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 Fig. 2(d). □

As mentioned by Chang et al. [4], the costs of computing the convex hull and managing the layer list such as inserting, deleting, and updating objects are not negligible. Inherently, we can adopt the maintenance method that was suggested by Chang et al. [4]. That is, since a layer is not affected by inserting or deleting objects inside the layer [4], we can reconstruct layers and their sorted lists over the objects only inside the layer. Due to the expensive computation cost, we suggest that the index reconstruction is performed in the batch mode as Chang et al. [4] do. Reducing the convex hull computation and incremental maintenance cost is not discussed in this paper since it is beyond the scope of this paper.

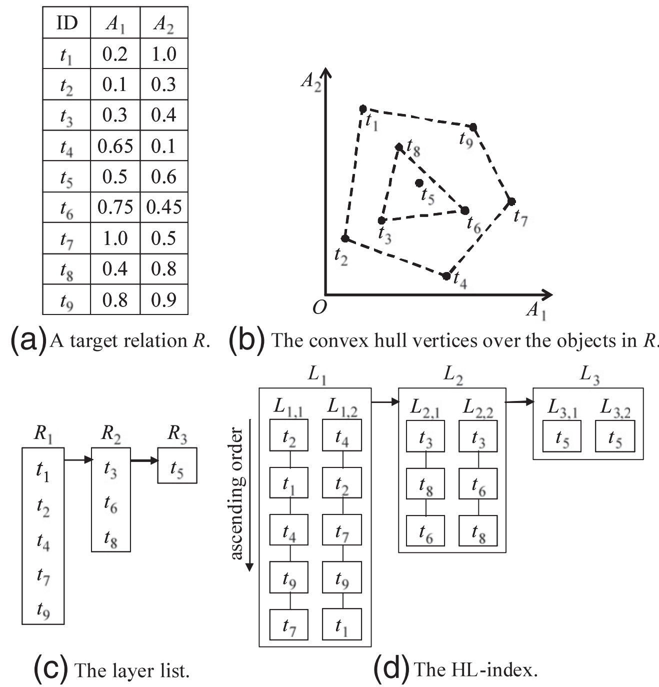
Before we proceed, we emphasize that, in our HL-index $L=\{L_1=\{L_{1,1},\ldots,L_{1,d}\},\ldots,L_m=\{L_{m,1},\ldots,L_{m,d}\}\}$, each sorted list $L_{i,j}$ contains only object identifiers, but not the full attribute values of the objects. We store their full attribute values in a separate place. 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. [34].

We now compute the storage cost of the HL-index. For ease of computing, we consider the storage cost of a numeric value as $\mathcal{O}(1)$. Let N_i be the number of objects in R_i. We note that each L_i contains d sorted lists, where each list has N_i object identifiers. For each L_i, to record the relationship between the layer and its sorted list, we maintain the identifiers of the d sorted lists. Then, the storage cost of L_i is computed as $\mathcal{O}(d*N_i+d)$. In addition, to maintain the sequence of the m layers, we use a list having m layer identifiers (i.e., layer numbers). Thus, the storage cost of the HL-index is computed as $\mathcal{O}(d*N+d*m+m)=\mathcal{O}(d*(N_1+\ldots+N_m)+d*m+m)$. Here, the first term (i.e., $\mathcal{O}(d*N)$) is equal to the storage cost of the list-based approach, and the sum of the third one (i.e., $\mathcal{O}(m)$) and the storage cost of N object identifiers $\mathcal{O}(N)$ is equal to the storage cost of the layer-based approach (i.e., $\mathcal{O}(N+m)$).

5. 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 5.1, we start reviewing the ONION algorithm [4] to explain how the HL-index can be used for layer-level filtering. Then in Section 5.2, we explain how we can extend the ONION algorithm to enable list-level filtering.

[1] Again, the HL-index for all monotone functions are described briefly in Section 5.5.



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

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

5.1. ONION algorithm: layer-level filtering

In the HL-index construction algorithm of Fig. 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. [4], 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. [4] proved the following important theorem.

Theorem 1. [4] *(Optimally Linearly Ordered Set Property) Let* $L=\{L_1,L_2,\ldots,L_m\}$ *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* $L_{i+1},\ldots,L_m$ *can have a score less than* $o_{min}(L_i)$*. That is,* $\forall o \in L_j\ (i<j\leq m)$, $f_{\bar{w}}(o_{min}(L_i)) \leq f_{\bar{w}}(o)$ *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. [4] proved the following corollary.

Corollary 1. [4] *Let* $o_{min}(L_i)$ *be the minimum-scored object in the layer* L_i*. Let* $H(i)$ *be the objects in* $L_1,L_2,\ldots,L_i$ *whose scores are* $f_{\bar{w}}(o_{min}(L_i))$ *or less. That is,* $H(i)=\{o|f_{\bar{w}}(o)\leq f_{\bar{w}}(o_{min}(L_i))\ \text{for}\ o\in L_1\cup L_2\cup\ldots\cup L_i\}$.[2] *If* $H(i)$ *contains k or more objects (i.e., if* $|H(i)\geq k|$*),* $H(i)$ *contains the top k objects.*

Based on Corollary 1, Chang et al. [4] proposed the ONION algorithm. In Fig. 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.

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 [4]. 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 "≥" instead of ">" [8], and for finding k objects with the lowest scores as the results, we use "≤".

[3] We present a slightly modified algorithm from what was proposed by Chang et al. [4] in order to make our later discussion easier to follow.

1. FOR $i = 1$ to m DO BEGIN
2. Evaluate $f_{\bar{w}}(o)$ for all $o \in L_i$
3. Find $o_{min}(L_i)$ from L_i where $f_{\bar{w}}(o_{min}(L_i)) \le f_{\bar{w}}(o)$ for any $o \in L_i$
4. Compute $H(i) = \{o \mid f_{\bar{w}}(o) \le f_{\bar{w}}(o_{min}(L_i))$ for $o \in L_1 \cup ... \cup L_i\}$
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.

5.2. List-level filtering for the 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. 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 Fig. 2(d). We set $S_i(n)$ to be $S_{i,1}(n)\cup S_{i,2}(n)\cup\ldots\cup S_{i,d}(n)$. Informally, $S_i(n)$ can be considered as the set of objects that we "see" by retrieving the first n objects from the head of each list $L_{i,j}$. For instance, S_1, $(2)=\{t_1,t_2,t_4\}$ in Fig. 2(d). We set $U_i(n)=L_i-S_i(n)$. For example, U_1 $(2)=L_1-S_1$ $(2)=\{t_1,t_2,t_4,t_7,t_9\}-\{t_1,t_2,t_4\}=\{t_7,t_9\}$ss. 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 the head of each list $L_{i,j}$. We use $a_{i,j}(n)$ to refer to the A_j attribute value of the nth object at the head of the list $L_{i,j}$. For example, in Fig. 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, $a_{i,j}$ (m) monotonically increases as n increases. Finally, we set $\mathcal{F}_i(n)=f(a_{i,1}(n),a_{i,2}(n),\ldots a_{i,d}(n))$. The meaning of the new set of symbols is summarized in Table 2. Under this notation, Fagin et al. [3] proved the following important theorem:

Theorem 2. [3] *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,*

$$\forall o \in U_i(n), \quad f(o) \ge f\left(a_{i,1}(n), a_{i,2}(n), \ldots, a_{i,d}(n)\right) = \mathcal{F}_i(n). \tag{2}$$

Now, we extend the notations to handle non-monotone linear functions. 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 $w[j]$'s sign, we cannot guarantee that the threshold value monotonically 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 monotonically increases. To ensure that $w[j]*t[j]$ monotonically increases as we access more objects, we use alternative definitions of $S_{i,j}(n)$ and $a_{i,j}(n)$ in Table 2 depending on the sign of $w[j]$. If $w[j]\ge 0$, we use the "head" versions of the definitions in Table 2. Otherwise, we use the "tail" versions. We call this updated access mechanism *monotone access*. Using this notation, we naturally have the following corollary:

Corollary 2. 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,

$$\forall o \in U_i(n), \quad f_{\bar{w}}(o) \ge f_{\bar{w}}\left(a_{i,1}(n), a_{i,2}(n), \ldots, a_{i,d}(n)\right) = \mathcal{F}_i(n).$$

Given Corollary 2, we can see that Corollary 3 in Section 5.2.1 and Corollary 4 in Section 5.2.2 are still true if we replace the "monotone linear function f" with the "linear function $f_{\bar{w}}$." We will call them modified Corollary 3 and modified Corollary 4, respectively.

Fig. 4 shows the function *getNextObjects*() for the monotone access. In Line 3, for each $L_{i,j}$ ($j \in SUB$), it checks the sign of the jth 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. Here, we assume that, since the zero-weight attributes do not affect the final object score, *getNextObjects*() needs to retrieve objects only from the *non-zero-weight attribute* lists.

Example 2. Let us assume that $w[1]=0.5$ and $w[2]=-0.5$. Then, the first time *getNextObjects*() is called on L_1 in Fig. 2(d), it returns the top object t_2 from the list $L_{1,1}$ and the bottom object t_1 from $L_{1,2}$. The second time *getNextObjects*() is called on L_1, it returns the next objects, t_1 and t_9. □

5.2.1. Identifying o_{min} (L_i)

Theorem 2 provides an important clue on how we can identify 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 $S_i(n)$, if o_{min} $(S_i(n))$ has

```
Function getNextObjects()
Input: (1) L_i: the d sorted list L_{i,1}, ..., L_{i,d} of L_i
       (2) SUB: the set of the sequence numbers of the attributes mentioned
               in the query
       (3) f_w(): the linear score function
Output: S: the set of the next objects of L_{i,j}'s (j ∈ SUB)
1.  S := {}
2.  FOR EACH j ∈ SUB DO BEGIN
3.     IF w[j] ≥ 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 ∪ {o}
9.  END /* FOR */
10. RETURN S
```

Fig. 4. A function for the *BasicLayerbasedThresholdAlgorithm*.

a score less than or equal to $\mathcal{F}_i(n)$, then omin $(S_i(n))$ is the minimum-scored object in L_i. More precisely, Fagin et al. [3] proved the following corollary.

Corollary 3. [3] *Let o_{min} $(S_i(n))$ be the minimum-scored object in $S_i(n)$. Under any monotone linear function f, if $f(o_{min}(S_i(n))) \leq \mathcal{F}_i(n)$, then $f(o_{min}(S_i(n))) = f(o_{min}(L_i))$.*

Based on Corollary 3, Fagin et al. [3] proposed the TA algorithm. In Fig. 5 we show a modified version of the TA algorithm identifying o_{min} (L_i) by retrieving the first few objects from each list $L_{i,j}$. Starting from $n = 1n = 1$, 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 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 3, we can see that the returned object is the minimum-scored object in L_i.

5.2.2. Identifying H(i)

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

Corollary 4. *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 $S_1(n_1) \cup S_2(n_2) \cup \ldots \cup S_i(n_i)$.*

Proof. Let o be an object in $H(i)$. From the definition of $H(i)$, $H(i)$ is a subset of $L_1 \cup \ldots \cup L_i$, 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 $S_j(n_j) \subseteq S = S_1(n_1) \cup \ldots \cup S_i(n_i)$, o must be in S. That is, all the objects in $H(i)$ exist in S. Thus, $H(i) \subseteq S$. □

From Corollary 4, we can compute the $H(i)$ by retrieving the first few objects in each list $L_{i,j}$ from the layers L_1 through L_i without retrieving all the objects in L_j. Now we are ready to introduce our algorithm that performs both layer-level filtering and list-level filtering using the HL-index.

5.3. Basic algorithm

Fig. 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 $Q = (SUB, f_{\vec{w}}(), k)$. The output is the k objects having the lowest scores for the

```
1. n := 0; S_i(n) := {}
2. DO BEGIN
3.     n := n+1
4.     S_i(n) := S_i(n) ∪ 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. 5. A modified TA algorithm identifying $o_{min}(L_i)$ from L_i.

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) ∪ getNextObjects(L_i, SUB, f_w())
6.      END WHILE (f_w(o_min(S_i(n_i))) > F_i(n_i))
7.      /* Retrieve more objects to compute H(i) */
8.      FOR l = 1 TO i - 1 DO BEGIN  /* If i ≥ 2, perform this loop */
9.          WHILE (F_l(n_l) ≤ f_w(o_min(S_i(n_i)))) DO BEGIN
10.             n_l := n_l + 1
11.             S_l(n_l) := S_l(n_l) ∪ getNextObjects(L_l, SUB, f_w())
12.         END  /* WHILE */
13.     END  /* FOR */
14.     /* Does H(i) contains k or more objects ? */
15.     IF (S_1(n_1) ∪ ... ∪ S_i(n_i) contains k or more objects whose scores are
                      f_w(o_min(S_i(n_i))) or lower) THEN
16.         RETURN the top-k objects in S_1(n_1) ∪ ... ∪ S_i(n_i)
17. END  /* FOR */
```

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

scoring function $f_{\bar{w}}()$. Starting from $i=1$, the algorithm first computes $o_{min}(L_i)$ in Lines 2 through 6 (like the modified TA in Fig. 5). 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 $S_1(n_1) \cup \ldots \cup S_i(n_i)$ 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 $S_1(n_1) \cup \ldots \cup S_i(n_i)$. Otherwise, it increases i by one and repeats the process. We now 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 $S_1(n_1) \cup \ldots \cup S_i(n_i)$ 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 3 that

$$f_{\bar{w}}(o_{min}(S_i(n_i))) = f_{\bar{w}}(o_{min}(L_i)). \tag{3}$$

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

(1) $o \in L_i$: From Eq. (3), $o_{min}(S_i(n_i))$ 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. (3), $f_{\bar{w}}(o) \geq f_{\bar{w}}(o_{min}(S_i(n_i)))$.
(3) $o \in L_l$ for $l < i$: Due to the condition in Line 9, n_l is increased until

$$\mathcal{F}_l(n_l) > f_{\bar{w}}(o_{min}(S_i(n_i))). \tag{4}$$

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 2, such an o satisfies

$$f_{\bar{w}}(o) \geq \mathcal{F}_l(n_l). \tag{5}$$

From Eqs. (4) and (5), we get $f_{\bar{w}}(o) > f_{\bar{w}}(o_{min}(S_i(n_i)))$. □

5.4. 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 $min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i)))) \le f_{\bar{w}}(o_{min}(L_i))$ 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 $\mathcal{F}_i(n_i) \le f_{\bar{w}}(o_{min}(L_i))$ from Corollary 2. Thus, $min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i)))) \le f_{\bar{w}}(o_{min}(L_i))$. □

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.

Fig. 7 shows *EnhancedLayerbasedThresholdAlgorithm* (simply, *EnhancedLTA*), which implements this idea. The inputs and the output of EnhancedLTA are same to those of BasicLTA in Fig. 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 does 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 exists 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 $S_1(n_1) \cup \ldots \cup S_i(n_i)$ at Line 15 satisfies $f_{\bar{w}}(o) \ge min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i))))$. Consider three cases of o in R but not in $S_1(n_1) \cup \ldots \cup S_i(n_i)$:

(1) $o \in L_i$: Since $o_{min}(L_i)$ is the minimum-scored object in L_i, $f_{\bar{w}}(o) \ge f_{\bar{w}}(o_{min}(L_i)) \ge min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i))))$ from Lemma 1.
(2) $o \in L_l$ for $l > i$: From Theorem 1, we know that $f_{\bar{w}}(o) \ge f_{\bar{w}}(o_{min}(L_i)) \ge min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i))))$.

```
Algorithm EnhancedLayerbasedThresholdAlgorithm(EnhancedLTA):
Input: (1) [L_1 = {L_1,1, …, L_1,d}, …, L_m = {L_m,1, …, L_m,d}]: an HL Index
       (2) Q = (SUB, f_w(), k): a subspace top-k query
Output: [top-1, …, top-k]: the sequence of k objects with the lowest scores
        for f_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) ∪ getNextObjects(L_i, SUB, f_w())
6.        /* Retrieve more objects */
7.        FOR l = 1 TO i - 1 DO BEGIN /* If i ≥ 2, perform this loop */
8.           WHILE (F_l(n_l) ≤ min(f_w(o_min(S_i(n_i))), F_i(n_i))) DO BEGIN
9.              n_l := n_l + 1
10.             S_l(n_l) := S_l(n_l) ∪ getNextObjects(L_l, SUB, f_w())
11.          END /* WHILE */
12.       END /* FOR */
13.       /* Does S_1(n_1) ∪ ... ∪ S_i(n_i) contains k or more objects ? */
14.       IF (S_1(n_1) ∪ ... ∪ S_i(n_i) contains k or more objects whose scores
                are min(f_w(o_min(S_i(n_i))), F_i(n_i)) or lower) THEN
15.          RETURN the top-k objects in S_1(n_1) ∪ ... ∪ S_i(n_i)
16.    END WHILE (f_w(o_min(S_i(n_i))) > F_i(n_i))
17. END /* FOR */
```

Fig. 7. The query processing algorithm enhanced to use a tighter bound.

(3) $o \in L_l$ for $l < i$: Due to the condition in Line 8, n_l is increased until

$$\mathcal{F}_l(n_l) > min(\mathcal{F}_i(n_i), f_{\bar{w}}(o_{min}(S_i(n_i)))). \tag{6}$$

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 2, 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. (6). □

5.5. HL-index for monotone functions

We now briefly explain how we can 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* [23] 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 5 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. 7 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, $L_1 \cup \ldots \cup L_m$ is the entire database, and L_i is the set of objects in the skyline over $L_i \cup \ldots \cup L_m$. According to the definition of the skyline [23],[4] when L_i is the skyline over $L_i \cup \ldots \cup L_m$, every object in $L_{i+1} \cup \ldots \cup L_m$ is dominated by at least one object in L_i. That is, for any $o \in L_{i+1} \cup \ldots \cup L_m$, 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,

$$\forall o \in L_{i+1} \cup \ldots \cup L_m, \quad \exists o' \in L_i \text{ s.t. } f(o') \leq f(o). \tag{7}$$

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. □

In addition, we can easily extend the HL-index to handle a monotone linear function only by using the convex skyline [8], which is computed from the skyline and the convex hull, as the layering scheme. This is because the convex skyline contains at least one object that minimizes an arbitrary monotone linear function, and the layer list consisting of convex skylines satisfies the optimally linearly ordered set property [8]. Thus, we can also use the identical algorithm in Fig. 7.

From now on, if we need to differentiate these two new versions of the HL-index from the earlier one, we refer to the earlier one as HL-index (convex), the first new one as HL-index (skyline), and the second new one as HL-index (convex skyline) (simply, HL-index(*cvxsky*)).

6. Experiments

6.1. Experimental data and environment

We compare the index building cost and the query performance of the HL-index with the following existing methods: ONION [4] (a layer-based method), TA [3] (a list-based method), PREFER [6] (a view-based method), and SUB-TOPK [21], and PL-index [8].

We use the number of bytes as the measure of the index storage cost, 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 [4,6,9] 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. [23] and correlation datasets (*CORRELATION*) by using the generator used by Buruno et al. [35]. The datasets consist of three-, five-, and seven-dimensional datasets of 10 K, 100 K, and 1000 K objects. For the real dataset, we use the regular season statistics[5] of the NBA players that play over 10 min per year from 1951 to 2007 (*NBA*). The dataset consists of 19,364 objects with seven attributes: minutes played, total points, field goals attempted, free throws attempted, total rebounds, total assists, and total personal fouls.

[4] "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." [23]

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

Table 3
Index storage cost of a real data (NBA and $d=7$) (byte).

N	ONION			TA(e)			HL-index		
	objectID	*layerID*	*overhead*	*objectID*	*layerID*	*overhead*	*objectID*	*layerID*	*overhead*
19,364	77,456	36	1.000465	542,192	0	7.0	542,192	288	7.003718

For the experiments, we have implemented HL-index (including convex hull, skyline, and convex 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[6] written in JAVA into C++. To compute convex hulls for HL-index (convex and convex skyline), ONION, and PL-index, we used the Qhull library [36]. To compute skylines for HL-index (skyline and convex skyline), we used the BNL algorithm [23]. We conducted all the experiments on a Pentium-4 2.0 GHz Linux PC with 1GBytes of main memory.

6.2. Index building cost

We first compare the index storage cost and the index building time of HL-index, ONION, and TA(e). For the index storage cost, we first show the results for the NBA data in Table 3. Then, we show the results for the UNIFORM data while varying the number of objects ($N=10$ K, 100 K, 1000 K) and the dimension ($d=3$, 5, 7) in Tables 4 and 5. We used 4 byte integer to store an object (or layer or list) identifier. In Tables 3, 4, and 5, *objectID* represents the storage cost of maintaining object identifiers; *layerID* the storage cost of maintaining the relationship between the layers and their sorted lists; *overhead* the ratio of the *base* storage cost to the *index* storage cost. Here, the base storage cost is defined as the storage cost of storing the object identifiers without duplications, and the index storage cost is computed as the sum of *objectID* and *layerID*.

From these results, we observe that the storage overhead of HL-index over TA(e) is negligible because the difference between the HL-index and TA(e) is only up to 0.16% of TA(e) for all ranges of N and d. However, since the HL-index and TA(e) have d duplicated object identifiers in their sorted lists, their storage overheads are larger than that of ONION by about d times. Interestingly, Table 5 shows that the layerID costs of HL-index and ONION decrease as d increases. More precisely, the number of the layers, m, decreases as d increases because the layer size (i.e., the number of convex hull vertices) increases as d does. That is, $m=61$ when $d=3$, $m=28$ when $d=5$, and $m=12$ when $d=7$. Since the average layer size is computed as $(\ln N)^{d-1}$ [37], the layer size will be equal to N when d is very large. Table 5 precisely reflects this trend.

For the index building time, we show the results for the UNIFORM data of the dimension $d=5$ while varying the number of objects ($N=10$ K, 100 K, 1000 K) in Table 6. 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 are close to what we observe from Table 6. Similarly, the index building times of the HL-index (skyline) (or the HL-index (cvxsky)) is close to the sum of the time for constructing the skyline (or the convex skyline) layers and the time for building the lists of TA(e).

6.3. 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, SUB-TOPK, and PL-index support *monotone* or *monotone linear* functions and thus cannot handle non-monotone linear functions. In this section, therefore, we only compare HL-index (that uses the convex hull layering), ONION, and TA(e). The comparison of all eight 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 3), the number of results k, N, d, and the correlation factor *cf*.[7] 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 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 ith attribute ($i \in SUB$) in the preference vector $\bar{w}$, from $\{-4,-3,-2,-1,1,2,3,4\}$, and normalize $w[i]$'s so that $\sum_{i \in SUB}|w[i]|=1$.

6.3.1. 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. Fig. 8 shows the query performance of HL-index and HL-index(basic) as s is varied from 1 to 3, where HL-index(basic)

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

[7] The correlation factor cf represents degrees of correlation that range between −0.1 and 1.0. When cf is zero, attributes are independent of one another. Higher value of cf represents positive correlation between the values of one attribute and those of the other attributes. If cf is 1.0, the values of all attributes are equal. In contrast, when cf<0, the values of one attributes are negatively correlated with those of the other attributes [35].

Table 4
Index storage cost as N is varied (UNIFORM and $d=5$) (byte).

N	ONION			TA(e)			HL-index		
	objectID	*layerID*	*overhead*	*objectID*	*layerID*	*overhead*	*objectID*	*layerID*	*overhead*
10 K	40,000	52	1.0013	200,000	0	5.0	200,000	312	5.0078
100 K	400,000	112	1.00028	2,000,000	0	5.0	2,000,000	672	5.00168
1000 K	4,000,000	240	1.00006	20,000,000	0	5.0	20,000,000	1440	5.00036

Table 5
Index storage cost as d is varied (UNIFORM and $N=100{,}000$) (byte).

d	ONION			TA(e)			HL-index		
	objectID	*layerID*	*overhead*	*objectID*	*layerID*	*overhead*	*objectID*	*layerID*	*overhead*
3	400,000	244	1.00061	1,200,000	0	3.0	1,200,000	976	3.00244
5	400,000	112	1.00028	2,000,000	0	5.0	2,000,000	672	5.00168
7	400,000	48	1.00012	2,800,000	0	7.0	2,800,000	384	7.00096

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.

6.3.2. Experiment 1: query performance as N is varied

Fig. 9 shows the query performance of the HL-index, ONION, and TA(e) as N is varied from 10 K to 1000 K. From the result, we observe that the 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 $N=1000$ K. Interestingly, we note that TA(e) performs quite well for a small N (it shows performance close to HL-index for $N=10$ K), 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 [37]). 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.

6.3.3. Experiment 2: query performance as s is varied

Fig. 10 shows the query performance of the 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 4, 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 $s\geq 2$, 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).

6.3.4. Experiment 3: query performance as k is varied

Fig. 11 shows the query performance of the HL-index, ONION, and TA(e) as k is varied from 1 to 100. The HL-index outperforms the other methods for the entire range of k. The HL-index improves by 2.7 to 3.2 times over ONION and by 1.6 to 5.0 times over TA(e).

Table 6
Index building time as N is varied (UNIFORM and $d=5$) (sec).

N	ONION	TA(e)	HL-index		HL-index (skyline)		HL-index (cvxsky)	
			Layering	Listing	Layering	Listing	Layering	Listing
10 K	9.14	0.24	9.14	0.23	1.33	0.24	9.12	0.37
100 K	152.70	2.89	152.70	2.65	146.99	2.46	491.15	3.94
1000 K	2500.61	35.23	2500.61	31.57	17283.95	39.36	78416.25	48.18

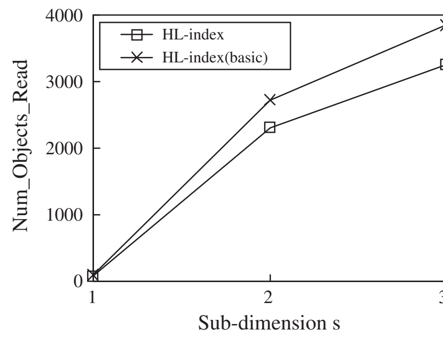


Fig. 8. Query performance of the basic and enhanced algorithms (NBA, $d = 7$, $N = 19{,}364$, and $k = 50$).

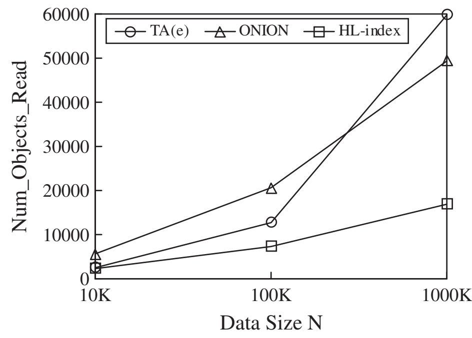


Fig. 9. Query performance as N is varied (UNIFORM, $d = 5$, $s = 3$, and $k = 50$).

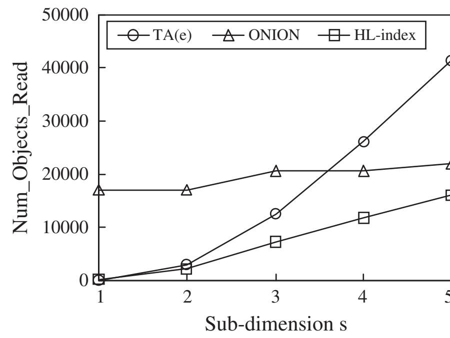


Fig. 10. Query performance as s is varied (UNIFORM, $d = 5$, $N = 100$ K, and $k = 50$).

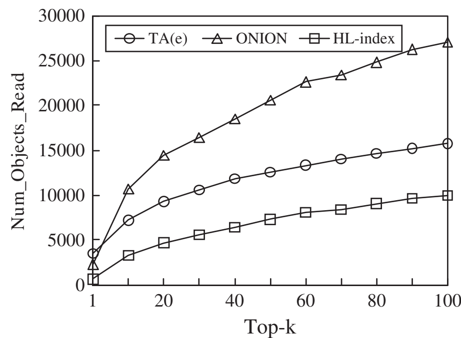


Fig. 11. Query performance as k is varied (UNIFORM, $d = 5$, $s = 3$, and $N = 100$ K).

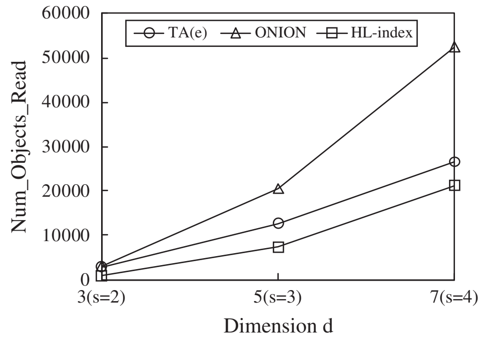


Fig. 12. Query performance as d is varied (UNIFORM, $N=100$ K, and $k=50$).

6.3.5. Experiment 4: query performance as d is varied

Fig. 12 shows the query performance of the 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 [37], the entire object can reside in the first layer when d is very large as we mentioned in Section 6.2. Thus, in that case, since the HL-index can exploit only the list-level filtering like TA(e), the performance of the HL-index would converge to that of TA(e). However, within the experimental range of d, the HL index outperforms TA(e). The HL-index improves by 2.5 to 3.2 times over ONION and by 1.2 to 3.0 times over TA(e).

6.3.6. Experiment 5: query performance as cf is varied

Fig. 13 shows the query performance of the HL-index, ONION, and TA(e) as *cf* is varied from −0.9 to 0.9. From the result, we observe that the query performance of the HL-index is not very sensitive to *cf* like ONION, but TA(e) is. This is because the HL-index and ONION exploit the relationship among the attributes *globally*, but TA(e) does *individually*. HL-index outperforms the other methods for all ranges of *cf* due to the synergic effect of the layer-level filtering and the list-level filtering. HL-index improves by 1.4 to 3.2 times over ONION and by 1.8 to 3.1 times over TA(e).

6.3.7. Experiment 6: query performance as s is varied when using a real dataset

Fig. 14 shows the query performance of the HL-index, ONION, and TA(e) as s is varied from 1 to 7 when using *NBA*, a seven-dimensional real dataset. The HL-index begins to outperform the other methods when $s \geq 2$. This tendency is similar to that

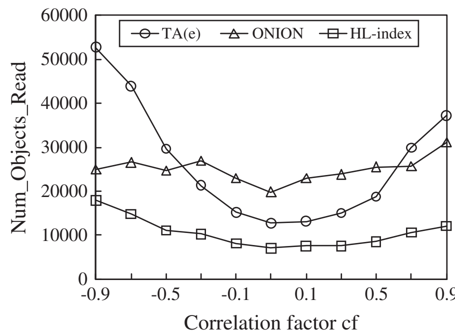


Fig. 13. Query performance as *cf* is varied (CORRELATION, $d=5$, $N=100$ K, $s=5$, and $k=50$).

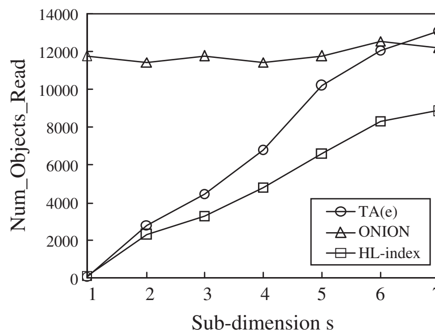


Fig. 14. Query performance as s is varied (NBA, $d=7$, $N=19{,}364$, and $k=50$).

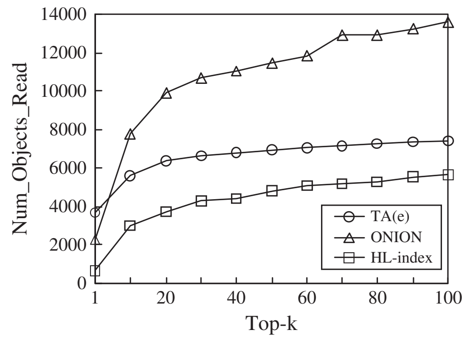


Fig. 15. Query performance as k is varied (NBA, $d=7$, $s=4$, $N=19{,}364$, and $k=50$).

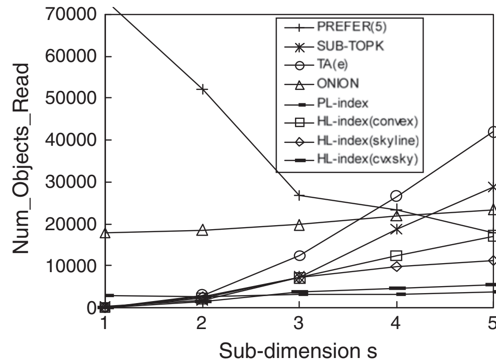


Fig. 16. Performance of monotone linear queries as s is varied (UNIFORM, $d=5$, $N=100$ K, and $k=50$).

of the synthetic dataset shown in Fig. 10. The HL-index improves by 1.4 to 138.9 times over ONION and by 0.6 to 1.6 times over TA(e).

6.3.8. Experiment 7: query performance as k is varied when using a real dataset

Fig. 15 shows the query performance of the HL-index, ONION, and TA(e) as k is varied from 1 to 100 when using NBA. The HL-index outperforms the other methods for the entire range of k. This tendency is similar to that of the synthetic dataset shown in Fig. 11. The HL-index improves by 2.3 to 3.3 times over ONION and by 1.3 to 5.3 times over TA(e).

6.4. Performance of monotone linear queries

We now compare the performance of all eight methods, HL-index (convex), HL-index (skyline), HL-index (cvxsky), ONION, TA(e), PREFER, SUB-TOPK, and the PL-index, 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 HL-index (cvxsky) and PL-index are designed to handle monotone linear functions, and cannot deal with non-monotone linear functions.

In generating the queries, we use the same setting that we used in Section 6.3, 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.

6.4.1. Experiment 8: query performance as s is varied when using only monotone linear scoring functions

Fig. 16 shows the query performance of HL-index (convex), HL-index (skyline), HL-index (cvxsky), ONION, TA(e), PREFER, SUB-TOPK, and PL-index as s is varied from 1 to 5 when using monotone linear scoring functions. 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 the HL-index, ONION, and TA(e) shows similar results to what we observed earlier; the HL-index shows significant improvement over ONION and TA(e) in almost all cases. HL-index (skyline) improves by up to 782.0 times over PREFER and by up to 2.6 times over SUB-TOPK. HL-index (cvxsky) improves by up to 41.3 over the PL-index when $s<3$. 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).

[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.

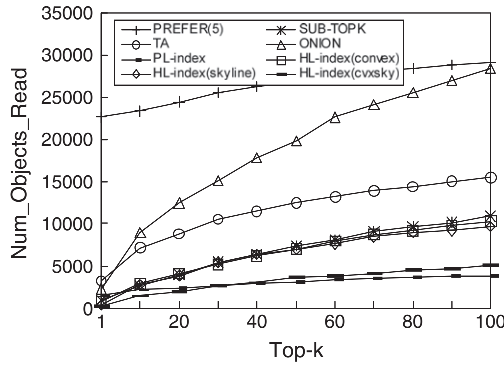


Fig. 17. Performance of monotone linear queries as k is varied (UNIFORM, $d=5$, $s=3$, and $N=100$ K).

6.4.2. Experiment 9: query performance as k is varied when using only monotone linear scoring functions

Fig. 17 shows the query performance of HL-index (convex), HL-index (skyline), HL-index (cvxsky), ONION, TA(e), PREFER, SUB-TOPK, and the PL-index as k is varied from 1 to 100 when using monotone linear scoring functions. Here again, PREFER uses five views. The performance trends of the HL-index, ONION, and TA(e) are similar to what we observed in earlier experiments; the HL-index shows significant improvement over ONION and TA(e). Compared to SUB-TOPK, HL-index (skyline) shows approximately 10% (which should be higher with a higher sub-dimension such as $s\geq4$) performance improvement in many cases. Compared to PREFER, HL-index (skyline) shows up to 55.6 times performance improvement. Compared to the PL-index, HL-index (cvxsky) shows up to 5.9 times performance improvement when $k\leq40$. 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.

6.4.3. Experiment 10: query performance as cf is varied when using only monotone linear scoring functions

Fig. 18 shows the query performance of HL-index (convex), HL-index (skyline), HL-index (cvxsky), ONION, TA(e), PREFER, SUB-TOPK, and the PL-index as *cf* is varied from −0.9 to 0.9 when using monotone linear scoring functions. Here again, PREFER uses five views. From the result, we observe that the query performances of HL-index and ONION are not very sensitive to *cf*, but TA(e) is. This trend is similar to that in Experiment 5. The HL-index shows significant improvement over ONION and TA(e). Compared to SUB-TOPK, HL-index (skyline) shows 2.0 times performance improvement. Compared to PREFER, HL-index (skyline) shows up to 19.6 times performance improvement. Compared to the PL-index, HL-index (cvxsky) shows up to 1.8 times performance improvement.

7. 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

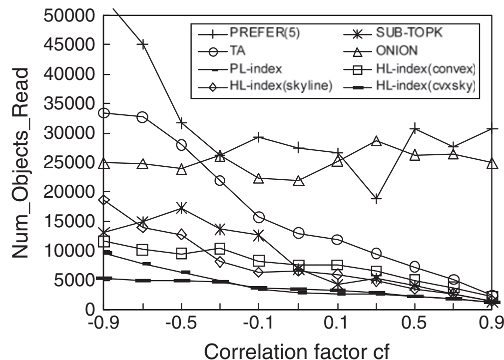


Fig. 18. Performance of monotone linear queries as *cf* is varied (CORRELATION, $d=5$, $N=100$ K, $s=5$, and $k=50$).

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 2^d, 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 [38].

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.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by Korean Government (MEST) (no. 2012R1A2A1A05026326). An earlier version [39] of this paper was presented at IEEE ICDE 2010 as a short paper. This paper extends the earlier one by adding an enhanced algorithm with a tighter bound and a new variation for non-linear monotone functions. Besides, it includes correctness proofs of the algorithms and extensive experiments.

References

[1] I.F. Ilyas, G. Beskales, M.A. Soliman, A survey of top-k query processing techniques in relational database systems, ACM Computing Surveys 40 (4) (2008).
[2] R. Fagin, Fuzzy queries in multimedia database systems, in: Proc. 20th ACM Symposium on Principles of Database Systems (PODS), Seattle, Washington, June 1998, 1998, pp. 1–10.
[3] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, in: Proc. 17th ACM Symposium on Principles of Database Systems (PODS), Santa Barbara, California, May 2001, 2001, pp. 102–113.
[4] 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: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, Dallas, Texas, May 2000, 2000, pp. 391–402.
[5] G. Das, D. Gunopulos, N. Koudas, D. Tsirogiannis, Answering top-k queries using views, in: Proc. 32nd Int'l Conf. on Very Large Data Bases (VLDB), Seoul, Korea, Sept. 2006, 2006, pp. 451–462.
[6] V. Hristidis, Y. Papakonstantinou, Algorithms and applications for answering ranked queries using ranked views, The VLDB Journal 13 (1) (2004) 49–70.
[7] K. Yi, H. Yu, J. Yang, G. Xia, Y. Chen, Efficient maintenance of materialized top-k views, in: Proc. 19th Int'l Conf. on Data Engineering (ICDE), Bangalore, India, Mar. 2003, 2003, pp. 189–200.
[8] J.-S. Heo, K.-Y. Whang, M.-S. Kim, Y.-R. Kim, I.-Y. Song, The partitioned-layer index: answering monotone top-k queries using the convex skyline and partitioning-merging technique, Information Sciences 179 (9) (2009) 3286–3308.
[9] D. Xin, C. Chen, J. Han, Towards robust indexing for ranked queries, in: Proc. 32nd Int'l Conf. on Very Large Data Bases (VLDB), Seoul, Korea, Sept. 2006, 2006, pp. 235–246.
[10] G. Hadley, Linear Programming, Addison-Wesley Publishing Company, 1962.
[11] M. Berg, M. Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry: Algorithms and Applications, 2nd ed. Springer-Verlag, 2000.
[12] S.G. Gass, Linear Programming: Method and Applications, 5th ed. An International Thomson Publishing Company, 1985.
[13] M. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum, IO-top-k: index-access optimized top-k query processing, in: Proc. 32nd Int'l Conf. on Very Large Data Bases (VLDB), Seoul, Korea, Sept. 2006, 2006, pp. 475–486.
[14] S. Nepal, M.V. Ramakrishna, Query processing issues in image (multimedia) databases, in: Proc. 15th Int'l Conf. on Data Engineering (ICDE), Sydney, Australia, Mar. 1999, 1999, pp. 22–29.
[15] R. Akabarina, E. Pacitti, P. Valduriez, Best position algorithms for top-k queries, in: Proc. Int'l Conf. on Very Large Data Bases (VLDB), Vienna, Austria, Sept. 2007, 2007, pp. 495–516.
[16] H. Pang, X. Ding, B. Zheng, Efficient processing of exact top-k queries over disk-resident sorted lists, The VLDB Journal 19 (3) (June 2010).
[17] C.-Y. Chan, P.-K. Eng, K.-L. Tan, Stratified computation of skylines with partially-ordered domains, in: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, Baltimore, Maryland, June 2005, 2005, pp. 203–214.
[18] D. Papadias, Y. Tao, G. Fu, B. Seeger, Progressive skyline computation in database systems, ACM Transactions on Database Systems 30 (1) (2005).
[19] W. Jin, J.M. Patel, Efficient and generic evaluation of ranked queries, in: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, Athens, Greece, June 2011, 2011, pp. 601–612.
[20] K.-L. Tan, P.-K. Eng, B.C. Ooi, Efficient progressive skyline computation, in: Proc. 27th Int'l Conf. on Very Large Data Bases (VLDB), Roma, Italy, Sept. 2001, 2001, pp. 301–310.
[21] Y. Tao, X. Xiao, J. Pei, Efficient skyline and top-k retrieval in subspaces, IEEE Transactions on Knowledge and Data Engineering 19 (8) (2007).
[22] S. Zhang, N. Mamoulis, D. Cheung, Scalable skyline computation using object-based space partitioning, in: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, Rhode Island, USA, June 2009, 2009, pp. 483–494.
[23] S. Borzsonyi, D. Kossmann, K. Stocker, The skyline operator, in: Proc. 17th Int'l Conf. on Data Engineering (ICDE), Heidelberg, Germany, Apr. 2001, 2001, pp. 421–430.
[24] G. Beskales, M.A. Soliman, I.F. Ilyas, Efficient search for the top-k probable nearest neighbors in uncertain databases, in: Proc. 34th Int'l Conf. on Very Large Data Bases (VLDB), Auckland, New Zealand, Aug. 2008, 2008, pp. 326–339.
[25] M. Hua, J. Pei, W. Zhang, X. Lin, Ranking queries on uncertain data: a probabilistic threshold approach, in: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, Vancouver, Canada, June 2008, 2008, pp. 673–686.
[26] L.A. Zadeh, Toward a generalized theory of uncertainty (GTU), Information Sciences 172 (1–2) (2005) 1–40.
[27] C. Wang, L.Y. Yuan, J.-H. You, O.R. Zaiane, J. Pei, On pruning for top-k ranking in uncertain databases, in: Proc. 37th Int'l Conf. on Very Large Data Bases (VLDB), Seattle, WA, USA, Aug. 2011, 2011, pp. 598–609.
[28] C. Cong, C. Jensen, D. Wu, Efficient retrieval of the top-k most relevant spatial web objects, in: Proc. 35th Int'l Conf. on Very Large Data Bases (VLDB), Lyon, France, Aug. 2009, 2009, pp. 337–348.
[29] S.-W. Hwang, K.C.-C. Chang, Optimizing top-k queries for middleware access: a unified cost-based approach, ACM Transactions on Database Systems (TODS) 32 (1) (2007).

[30] K. Zhao, Y. Tao, S. Zhou, Efficient top-k processing in large-scaled distributed environments, Data & Knowledge Engineering 63 (2) (2007) 315–335.
[31] A. Vlachou, C. Doulkeridis, Y. Kotidis, K. Norvag, Reverse top-k queries, in: Proc. 26th Int'l Conf. on Data Engineering (ICDE), Long Beach, California, Mar. 2010, 2010, pp. 365–376.
[32] A. Yu, P.K. Agrawal, J. Yang, Processing a large number of continuous preference top-k queries, in: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, Scottsdale, Arizona, USA, May 2012, 2012, pp. 397–408.
[33] N. Bruno, S. Chaudhuri, L. Gravano, Top-k selection queries over relational databases: mapping strategies and performance evaluation, ACM Transactions on Database Systems (TODS) 27 (2) (2002).
[34] S. Chaudhuri, R. Ramakrishnan, G. Weikum, Integrating DB and IR technologies: what is the sound of one hand clapping? in: Proc. Second Biennial Conference on Innovative Data Systems Research, CIDR2005, Asilomar, California, Jan. 2005, 2005, pp. 1–12.
[35] N. Bruno, L. Gravano, A. Marian, Evaluating top-k queries over web-accessible databases, in: Proc. 26th Int'l Conf. on Data Engineering (ICDE), Long Beach, San Jose, CA, Mar. 2002, 2002, pp. 369–380.
[36] B. Barber, D. Dobkin, H. Huhdanpaa, The Quickhull algorithm for convex hulls, ACM Transactions on Mathematical Software 22 (4) (1996).
[37] J.L. Bentley, H.T. Kung, M. Schkolnick, C.D. Thompson, On the average number of maxima in a set of vectors and applications, Journal of the ACM 33 (2) (1978).
[38] S.-W. Lee, B. Moon, C. Park, J.M. Kim, S.-W. Kim, A case for flash memory SSD in enterprise database applications, in: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, Vancouver, Canada, June 2008, 2008, pp. 1075–1086.
[39] J.-S. Heo, J. Cho, K.-W. Whang, The hybrid-layer index: a synergic approach to answering top-k queries in arbitrary subspaces, in: Proc. 26th Int'l Conf. on Data Engineering (ICDE), Long Beach, California, USA, Mar. 2010, 2010, pp. 445–448.

Jun-Seok Heo received the B.S. (1995) and M.S. (1997) degrees in Computer Science and Statistics from University of Seoul. He earned a Ph.D. (2009) degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST). From 1997 to 2002, he was a senior researcher at Daewoo Telecom Co., Ltd. and Mercury Co. In 2002, he was a research staff at the Advanced Information Research Center (AITrc). From 2009 to 2010, he was a post-doctoral researcher in KAIST. Currently, he is working in SAP Labs Korea, Inc. His research interests include information retrieval, sensor networks, geographic information systems, and main-memory database systems.

Junghoo Cho is an associate professor in the Department of Computer Science at University of California, Los Angeles. He received a Ph.D. degree in Computer Science from Stanford University and a B.S. degree in physics from Seoul National University. His main research interests are in the study of the evolution, management, retrieval and mining of information on the World-Wide Web. He publishes research papers in major international journals and conference proceedings. He serves on program committees of top international conferences, including SIGMOD, VLDB and WWW. He is a recipient of the 10-Year Best-Paper Award at VLDB 2010, NSF CAREER Award, IBM Faculty Award, Okawa Research Award and Northrop Grunmann Excellence in Teaching Award.

Kyu-Young Whang graduated (Summa Cum Laude) from Seoul National University in 1973 and received an M.S. degree from Korea Advanced Institute of Science and Technology (KAIST) in 1975, and Stanford University in 1982. He earned a Ph.D. degree from Stanford University in 1984. From 1983 to 1991, he was a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, NY. In 1990, he joined KAIST, where he currently is a KAIST Distinguished Professor at the Department of Computer Science and the Director of the Advanced Information Technology Research Center (AITrc). His research interests encompass database systems/storage systems, search engines, object-oriented databases, multimedia databases, geographic information systems (GIS), data mining/data warehouses, XML databases, and data streaming.

Dr. Whang served as the program co-chair of COOPIS'98, VLDB 2000, and ICDE2006. He was the general chair of VLDB2006, PAKDD 2003, and DASFAA 2004. Dr. Whang served as an Editor-in-Chief of the VLDB Journal from 2003 to 2009 (Coordinating EIC from 2007 to 2009) after serving the editorial board as a founding member for 13 years. He served as an associate editor of many journals including the IEEE Data Engineering Bulletin, Distributed and Parallel Databases, Int'l J. of GIS, and IEEE TKDE. He is currently on the editorial board of the WWW Journal. He was a trustee of the VLDB Endowment from 1998 to 2004 and was reelected in 2010. He served as Chair, Vice Chair, Advisor, and is currently serving as Awards Committee Chair of the steering committee of the DASFAA conference. He is on the steering committee of the IEEE ICDE, PAKDD, APWeb, and ADMA Conferences.

Dr. Whang is a Fellow of ACM and IEEE.