Toward Scalable Indexing for Top-k Queries

Jongwuk Lee, Hyunsouk Cho, Sunyou Lee, and Seung-won Hwang

Abstract—A top-k query retrieves the best k tuples by assigning scores for each tuple in a target relation with respect to a user-specific scoring function. This paper studies the problem of constructing an indexing structure for supporting top-k queries over varying scoring functions and retrieval sizes. The existing research efforts can be categorized into three approaches: *list-*, *layer-*, and *view-based* approaches. In this paper, we mainly focus on the layer-based approach that pre-materializes tuples into consecutive multiple layers. We first propose a *dual-resolution* layer that consists of *coarse-level* and *fine-level* layers. Specifically, we build coarse-level layers using *skylines*, and divide each coarse-level layer into fine-level sublayers using *convex skylines*. To make our proposed dual-resolution layer *scalable*, we then address the following optimization directions: 1) index construction; 2) disk-based storage scheme; 3) the design of the virtual layer; and 4) index maintenance for tuple updates. Our evaluation results show that our proposed method is more scalable than the state-of-the-art methods.

Index Terms—skyline, convex skyline, ∀-dominance, ∃-dominance, dual-resolution layer

1 Introduction

With the exponential growth in database sizes, fighting the flood of information is of critical importance in modern database systems. Toward this goal, *top-k queries* (or *ranked queries*) have gained considerable attention as an effective means for narrowing down the overwhelming amount of data. Specifically, a top-k query consists of a *retrieval size* k and a *scoring function* $\mathcal{F}$. The top-k query retrieves the best k tuples by assigning a numerical score for each tuple in a target relation $\mathcal{R}$ with respect to $\mathcal{F}$. A user specifies a *user preference* by adjusting a *weight* w (or *importance*) of attributes for $\mathcal{F}$, *e.g.*, the price is twice as important as the distance from the airport. In this paper, we assume that the user preference is represented by a *linear combination function*, which is widely used in the literature.

Fig. 1 illustrates a toy dataset stored in Hotel database systems. Given two criteria, *price* and *distance* from the airport, a user searches for the *top-5* hotels satisfying a user preference. In a geometric view, when the user preference is represented by a weight vector $w = (0.5, 0.5)$ (an arrow), we can identify top-5 tuples $\{a, b, f, d, e\}$, the scores of which can be computed by sweeping the perpendicular line (dashed), *e.g.*, $\mathcal{F}(a) = 3.5$.

To support efficient top-k query processing, existing algorithms exploit *pre-materialized* structures to avoid unnecessary access in $\mathcal{R}$. These can be categorized into the following three approaches:

- **Layer-based approach:** This builds consecutive layers as a global index structure on all attributes $\mathcal{A}$ [2]–[6]. Each tuple in $\mathcal{R}$ is associated with a certain layer by one-to-one mapping. This approach then traverses one layer at a time by leveraging the relationships between adjacent layers.
- **List-based approach:** This employs *sorted* lists for each attribute [7]–[15]. Given a scoring function $\mathcal{F}$, it accesses sorted lists in a round-robin manner and aggregates the values identified from each list until the best k tuples are found.
- **View-based approach:** This employs pre-computed top-k queries as *views* [16], [17]. Given a new top-k query, it finds the most similar materialized top-k query, and reuses its computation.

Unlike list- and view-based approaches that employ the existing indices in database systems, the layer-based approach is specifically designed for top-k query computation and thus is known to be more efficient than the others [4]–[6]. In this paper, we focus on optimizing the layer-based approach which guarantees that the first k layers include top-k answers regardless of scoring functions. In particular, to further reduce the number of accessed tuples in the k layers, we observed optimization room for tightening the relationships between tuples in adjacent layers at a finer granularity.

Toward this goal, we propose *a dual-resolution layer*. Specifically, it consists of two-level layers such as *coarse-level* layers and *fine-level* layers. The coarse-level layers are built by skylines, and each coarse-level layer is split into *fine-level* sublayers as *convex skylines*. This structure can reduce the number of accessed tuples by exploiting the *dominance* relationship between coarse-level layers as well as the *relaxed dominance* relationship between fine-level sublayers.

To make the proposed layer-based structure more scalable, we extend our preliminary work [1] in the following directions: (1) For the scenario where a maximum retrieval size is known beforehand, we propose

- *The authors are with the Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH), Pohang 790-784, Korea.*
E-mail: {julee, prory, sylque, swhwang}@postech.ac.kr.

Manuscript received 27 Sep. 2012; revised 19 July 2013; accepted 13 Aug. 2013. Date of publication 28 Aug. 2013; date of current version 29 Oct. 2014.
Recommended for acceptance by Y. Tao.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier 10.1109/TKDE.2013.149

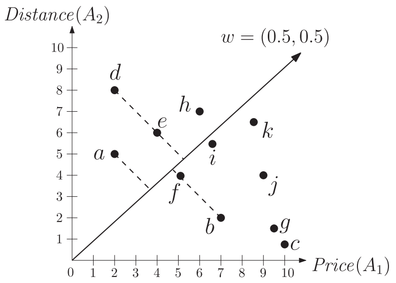
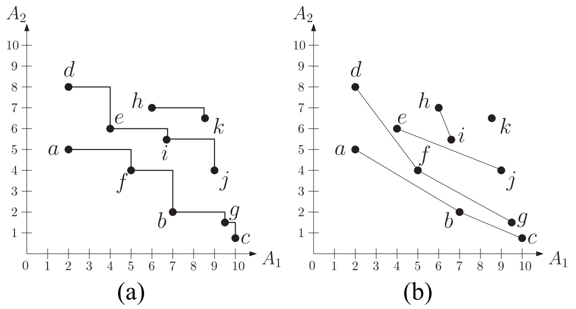


Fig. 1. Toy dataset in 2-D space.



(a) (b)

Fig. 2. Description of the layer-based approaches. (a) Skyline layers. (b) Convex layers.

optimization techniques to further prune out the tuples that cannot be the top-k answers. (2) When performing top-k query processing, we optimize for both the number of accessed tuples and I/O cost. (3) The first layer is further optimized for supporting more selective access. (4) When tuples are inserted or deleted, we amortize the update cost by trading retrieval efficiency with maintenance overhead.

In short, this paper makes the following contributions:

- We propose a dual-resolution layer that consists of coarse-level layers representing the skylines and fine-level layers representing the convex skylines (Section 3.1).
- We devise *static* pruning techniques for optimizing the construction of the dual-resolution layer by removing unnecessary tuples (Section 3.2).
- We develop efficient top-k query processing by traversing the dual-resolution layer through the relationships between tuples (Section 4).
- We design a disk storage scheme to reduce both the number of accessed tuples and the I/O cost during top-k query processing (Section 5).
- We optimize the selective access on the first layer using the *proximity* of top-1 candidate tuples in a systematic manner (Section 6).
- We propose efficient maintenance methods for insertion and deletion operations in dynamic data environments (Section 7).
- We evaluate the scalability of our proposed method over extensive synthetic datasets and real-life datasets (Section 8).

2 Preliminaries

We first introduce basic notations to formalize the problem of top-k queries. Let $\mathcal{R} = \{t^1, \ldots, t^n\}$ be a target relation with d attributes $\mathcal{A} = (A_1, \ldots, A_d)$. A domain $dom(A_i)$ on each attribute is non-negative real values. A tuple $t \in \mathcal{R}$ is represented by $(t_1, \ldots, t_d)$ such that $\forall i \in [1, d] : t_i \in dom(A_i)$.

A top-k query consists of a scoring function $\mathcal{F}$ and a retrieval size k. We assume that the scoring function is a *linear combination function*, *i.e.*, $\mathcal{F}(t) = \sum_{i=1}^{d} w_i t_i$, where $w = (w_1, \ldots w_d)$ is a user-specific weight vector. Without loss of generality, the values of the weight vector are normalized to [0, 1] and $\sum_{i=1}^{d} w_i = 1$. The scoring function $\mathcal{F}$ thus preserves *monotonicity* [7], [10]. Given two tuples t and t', if $t_i \leq t'_i$ for $\forall A_i \in \mathcal{A}$, then $\mathcal{F}(t) \leq \mathcal{F}(t')$. Recall that this assumption is commonly used in the literature [2]–[5], [18]. Depending on user preferences, top-k queries can return k tuples with the highest or the lowest scores. In the rest of this paper, we assume that the k tuples with the lowest scores are returned as the top-k answers. We assume that ties are broken arbitrarily. The highest scoring tuples can be found by changing the sign of tuples.

Definition 1 (Top-k query). *Given a scoring function $\mathcal{F}$ and a retrieval size k, a* top-k query *returns an ordered set of k tuples $\{t^1, \ldots, t^k\}$ such that $\mathcal{F}(t^1) \leq \cdots \leq \mathcal{F}(t^k) \leq \mathcal{F}(t^l)$ and $t^l \in \mathcal{R} - \{t^i, \ldots t^k\}$.*

A key challenge for computing top-k queries is to reduce the number of tuples accessed as much as possible. To address this problem, existing work has exploited *pre-computed* indexing structures to avoid unnecessary data access. These can be categorized into three approaches: (1) *list-based*, (2) *layer-based*, and (3) *view-based* approaches. The layer-based approach builds upon the specialized structure for top-k query processing. (Section 9 reviews the pros and cons for the existing approaches in details.) In this paper, we focus on the layer-based approach that consists of consecutive multiple layers (Fig. 2). Let $\mathcal{L} = \{L^1, \ldots L^l\}$ be a set of layers. Each tuple in $\mathcal{R}$ belongs to a layer with a one-to-one mapping, *i.e.*, $L^i \cap L^j = \emptyset$ $(i \neq j)$ and $\bigcup_{i=1}^{l} L^i = \mathcal{R}$.

We next define two important notions used in the existing layer-based approaches. First, a *skyline* [19], [20] is used for identifying a set of top-1 candidates without requiring specific scoring functions. The skyline is based on the notion of *dominance*. A tuple t *dominates* another tuple t' if t is no worse than t' in all attributes.

Definition 2 (Dominance). *Given t and t', t* dominates *t' on $\mathcal{A}$, denoted $t \prec t'$, if $\forall$ $i \in [1, d]$: $t_i \leq t'_i$ and $\exists$ $j \in [1, d]$: $t_j < t'_j$.*

Definition 3 (Skyline). *A tuple t is a* skyline tuple *if any other tuple t' $(\neq t)$ does not dominate t on $\mathcal{A}$. The skyline is a set of skyline tuples, i.e.,* $\mathrm{SKY}(\mathcal{R}) = \{t \in \mathcal{R} | \nexists t' \in \mathcal{R} : t' \prec t\}$.

Skyline layers [4], [6] can be built by identifying the skylines in a sequential manner. Specifically, the first layer L^1 is the skyline of $\mathcal{R}$, and the i^{th} layer L^i $(i > 1)$ is the skyline of $\mathcal{R} - \bigcup_{l=1}^{i-1} L^l$. For example, Fig. 2(a) depicts the skyline layers using a toy dataset (Fig. 1). These consist of three layers, $L^1 = \{a, b, c, f, g\}$, $L^2 = \{d, e, i, j\}$, and $L^3 = \{h, k\}$.

Alternatively, a *convex skyline* [21] can be used. A convex skyline is a subset of a *convex hull* [22], which is the smallest convex polyhedron including all the tuples in $\mathcal{R}$. Formally, we define the convex skyline as follows:

Definition 4 (Convex skyline). *A tuple t is a* convex skyline tuple *if t has the minimum score for any linear combination function. The convex skyline is a set of convex skyline tuples, i.e.,* $\mathrm{CSKY}(\mathcal{R}) = \{t^* \in \mathcal{R} | t^* = argmin_{t \in R} \sum_{i=1}^{d} w_i t_i, w_i \in \mathbb{R}^+, \sum_{i=1}^{d} w_i = 1\}$.

Similar to the skyline layers, we can build the *convex layers* [2], [3], [23] by sequentially identifying the convex skylines. The first layer is the convex skyline of $\mathcal{R}$, and the i^{th} ($i > 1$) layer is the convex skyline of $\mathcal{R} - \bigcup_{l=1}^{i-1} L^l$. Fig. 2(b) depicts the convex skyline layers using the toy dataset (Fig. 1). These consist of five layers, $L^1 = \{a, b, c\}$, $L^2 = \{d, f, g\}$, $L^3 = \{e, j\}$, $L^4 = \{h, i\}$, and $L^5 = \{k\}$.

3 Proposed Indexing Structure

In this section, we first propose a *dual-resolution* layer which constitutes two-level layers for tightening the relationships between tuples at a finer granularity (Section 3.1). Note that defining the relationships between tuples is essential for reducing the number of accessed tuples in top-k query computation as much as possible. We then explain how to optimize the construction of the dual-resolution layer (Section 3.2).

3.1 Dual-Resolution Layer

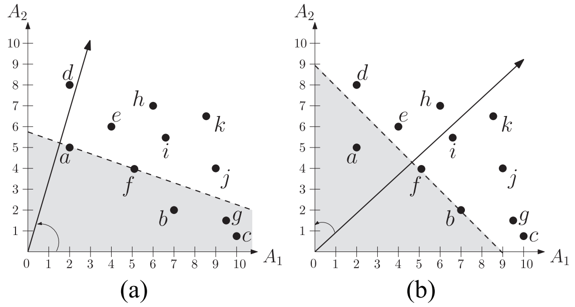
We build the dual-resolution layer with the following two phases:

1) **Coarse-level layer construction:** We build skyline layers sequentially. The first layer L^1 is the skyline $\mathrm{SKY}(\mathcal{R})$, and the other layer L^i ($i > 1$) is the skylines $\mathrm{SKY}(\mathcal{R} - \bigcup_{l=1}^{i-1} L^l)$.
2) **Fine-level layer construction:** We split the coarse-level layer into consecutive convex layers at a finer granularity. Let L^{ij} be the j^{th} fine-level layer in the L^i coarse-level layer. The first fine-level layer L^{i1} in L^i is the convex skyline $\mathrm{CSKY}(L^i)$, and the other fine-level layer L^{ij} is the convex skyline $\mathrm{CSKY}(L^i - \bigcup_{l=1}^{j-1} L^{il})$.

When the dual-resolution layer is built up, each tuple in $\mathcal{R}$ is associated with a certain layer. Given a scoring function $\mathcal{F}$ and a retrieval size k, the top-k answers are identified by traversing the layer through the relationships between tuples. In this process, some tuples are selectively accessed as the candidate tuples of the top-k answers. (Section 4 explains how to identify the top-k answers over the dual-resolution layer).

We now explain the relationships between tuples in adjacent layers over the dual-resolution layer. First, the dominance relationship for coarse-level layers can be used in the *dominant graph*, called DG [4], [6]. When sequentially accessing tuples from the i-th to the $(i+1)$-th coarse-level layers, we can skip the access of a tuple $t' \in L^{i+1}$ ($i > 1$) if every tuple $t \in L^i$ dominating t' is not accessed. We formally state this property. (Due to the space limitation, we omit all proofs and see [1]).

Lemma 1 (Dominance between coarse layers [4]). *For any scoring function $\mathcal{F}$, if $t \in L^i$ dominates $t' \in L^{i+1}$, the score $\mathcal{F}(t)$ of t is always smaller than the score $\mathcal{F}(t')$ of t'.*



(a) (b)

Fig. 3. Geometric view of the relationship between tuples. (a) $\mathcal{F}(a) \leq \mathcal{F}(f)$. (b) $\mathcal{F}(b) \leq \mathcal{F}(f)$.

Second, a relaxed dominance relationship between adjacent fine-level layers can be used for further selective access. Specifically, the coarse-level layer is divided into multiple fine-level layers. In this case, because no dominance relationship holds for adjacent fine-level layers, we adopt the relaxed dominance relationship for the fine-level layers. To highlight the different relationships, we alternatively call the classical definition of dominance $\forall$-*dominance*, and define a new dominance notion as $\exists$-*dominance*.

The next question is how to define $\exists$-dominance. From a geometric viewpoint, if a tuple $t \in L^{ij}$ has a smaller score than another tuple $t' \in L^{i(j+1)}$, the half-space of the perpendicular hyperplane on t' contains t. For instance, tuple f has a greater score than tuples a and b in each case (Fig. 3). By shifting the weight vectors, we can identify the tuple-wise relationship by comparing the scores of tuples. No matter how the weight vector is adjusted, a or b is always in the half-space of the perpendicular line lying on f (gray colors in Fig. 3). That is, either a or b always has a smaller score than f.

On the basis of this observation, we introduce an $\exists$-*dominance set* ($\mathcal{EDS}$). We consider a *hyperplane* $\mathcal{H}$ spanning a set of d tuples in d-dimensional space. Given a hyperplane $\mathcal{H}$ and a tuple t', it is said that the tuple set on hyperplane $\mathcal{H}$ is an $\mathcal{EDS}$ of tuple t' if $\mathcal{H}$ is closer to the origin than tuple t'. It is said that every tuple $t \in \mathcal{EDS}$ $\exists$-*dominates* t'. For example, a tuple set $\{a, b\}$ is represented by a line in two-dimensional space. When the line through $\{a, b\}$ is below tuple f, $\{a, b\}$ is an $\mathcal{EDS}$ of f, and a and b $\exists$-dominate f.

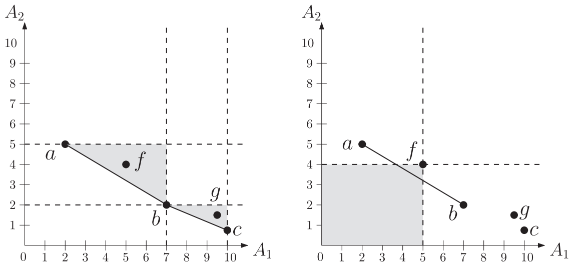
We formally define the $\exists$-*dominance set* and $\exists$-*dominance*. In addition, we state the $\exists$-dominance relationship between tuples in adjacent fine-level layers.

Definition 5 ($\exists$-dominance set). *A set of d tuples is an $\exists$-dominance set ($\mathcal{EDS}$) of a tuple t' on $\mathcal{A}$, i.e., $\mathcal{EDS}(t') = \{t^1, \ldots, t^d\}$, if a virtual tuple t^V on hyperplane $\mathcal{H}$ spanning the $\mathcal{EDS}$ dominates t'.*

Definition 6 ($\exists$-dominance). *Given two tuples t and t', t $\exists$-dominates t' on $\mathcal{A}$, denoted $t \precsim t'$, if $t \in \mathcal{EDS}(t')$.*

Lemma 2 ($\exists$-dominance between fine layers). *For any scoring function $\mathcal{F}$, if $t \in L^{ij}$ $\exists$-dominates $t' \in L^{i(j+1)}$, the score $\mathcal{F}(t)$ of t is smaller than the score $\mathcal{F}(t')$ of t.*

We now discuss how to find $\exists$-dominance sets to construct fine-level layers. Basically, the $\exists$-dominance sets for L^{ij} have to encompass all tuples in $L^{i(j+1)}$, *i.e.*, every tuple in $L^{i(j+1)}$ is connected by the $\exists$-dominance relationship for any tuple in L^{ij}. Otherwise, we might fail to identify correct



(a) (b)

Fig. 4. Relationships between tuples for fine-level layers. (a) Facets $\{a, b\}$ and $\{b, c\}$. (b) ∃-dominance for f.

top-k answers, as some tuples in $L^{i(j+1)}$ cannot be accessed from L^{ij}.

One naive method for identifying ∃-dominance sets is to enumerate all possible tuple sets $\binom{|L^{ij}|}{d}$, where $|L^{ij}|$ denotes the cardinality of L^{ij}. The complexity of this method is $O(|L^{ij}|^d)$. Because the number of dominance sets increases exponentially with the number of attributes, this incurs prohibitive computation cost.

To resolve this problem, we employ a set of *facets* which is a basic data structure representing a convex polyhedron [22]. Given a set of tuples in L^{ij}, it can be represented by a set of facets $\mathcal{FA} = \{FA^1, \ldots, FA^r\}$. Each facet is represented by a *hyperplane segment* that consists of a set of d adjacent tuples on $\mathcal{A}$. Fig. 4(a) describes a set of facets $\{\{a, b\}, \{b, c\}, \{a, c\}\}$ with respect to $\{a, b, c\}$. Due to the inherent property of the facet set, the facets can encompass all tuples in $L^{i(j+1)}$. This can be viewed as *minimal* ∃-dominance sets. In particular, because two facets $\{\{a, b\}, \{b, c\}\}$ are enough to cover $\{a, b, c\}$, the outside facet $\{a, c\}$ can be ignored. Therefore, given $\{\{a, b\}, \{b, c\}\}$, we check whether a facet can be an EDS of tuples in $L^{i(j+1)}$.

Example 1 (Relationships for fine-level layers). *Consider the first coarse-level layer in the toy dataset (Fig. 4). The coarse-level layer is split into two fine-level layers, $L^{11} = \{a, b, c\}$ and $L^{12} = \{f, g\}$. Two facets $\{a, b\}$ and $\{b, c\}$ in L^{11} (solid lines in Fig. 4(a) are regarded as ∃-dominance sets. Because the facets are represented by line segments, the subregions (i.e., gray triangles in Fig. 4(a)) are used for checking if $\{a, b\}$ or $\{b, c\}$ becomes an EDS of f or g. Because the line segment constituting $\{a, b\}$ goes through the dominating region of f (gray rectangle in Fig. 4(b)), a virtual tuple t^V on the segment dominates f. In this way, we can identify that $\{a, b\}$ ∃-dominates f and $\{b, c\}$ ∃-dominates g.*

We lastly explain the overall procedure for constructing a dual-resolution layer (Algorithm 1). First, we compute the skylines for the coarse-level layers (line 6). For each coarse-level layer, we then compute convex skylines as the fine-level layers and form the ∃-dominance relationships between fine-level layers (lines 7–16). After updating the fine-level layers, we also update the ∀-dominance relationships between coarse-level layers (lines 17–19). This process is iterated until every tuple in $\mathcal{R}$ is associated with a certain layer.

Example 2 (Dual-resolution layer construction). *Fig. 5 depicts a dual-resolution layer using a toy dataset (Fig. 1). Conceptually, the entire dataset is partitioned into three coarse-level layers (solid boxes), $L^1 = \{a, b, c, f, g\}$, $L^2 = \{d, e, i, j\}$, and $L^3 = \{h, k\}$. We update the ∀-dominance relationships between adjacent coarse-level layers (solid arrows), e.g., a ∀-dominates $\{d, e, i\}$. Each coarse-level layer is then divided into multiple fine-level layers (dotted boxes), $L^1 = \{\{a, b, c\}, \{f, g\}\}$, $L^2 = \{\{d, e, j\}, \{i\}\}$, and $L^3 = \{\{h, k\}\}$. The ∃-dominance relationships (dotted arrows) between fine-level layers are also updated, e.g., $\{b, c\}$ ∃-dominates g.*

Algorithm 1 BuildDualResolutionLayer($\mathcal{R}$)

```
Input: $\mathcal{R}$: a target relation on $\mathcal{A}$
Output: $\mathcal{L}$: a dual-resolution layer
1: $\mathcal{L} \leftarrow \{\}$ // Initialize a dual-resolution layer.
2: $i \leftarrow 1$ // Initialize the identifer of the coarse-level layer.
3: while $\mathcal{R} \neq \{\}$ do
4:     $L^i \leftarrow \{\}$ // Initialize the $i$-th coarse-level layer.
5:     $j \leftarrow 1$ // Initialize the identifier of the fine-level layer.
6:     $\mathcal{S} \leftarrow \mathrm{SKY}(\mathcal{R})$ // Compute a skyline for $\mathcal{R}$.
7:     while $\mathcal{S} \neq \{\}$ do
8:         // Compute the $j$-th fine-level layer in $L^i$.
9:         $L^{ij} \leftarrow \mathrm{CSKY}(\mathcal{S})$ // Compute a convex skyline for $\mathcal{S}$.
10:        if $j > 1$ then
11:            Update the ∃-dominance between $L^{i(j-1)}$ and $L^{ij}$
12:        end if
13:        $L^i \leftarrow L^i \cup L^{ij}$ // Insert $L^{ij}$ into $L^i$.
14:        $\mathcal{S} \leftarrow \mathcal{S} - L^{ij}$ // Update current skyline $\mathcal{S}$.
15:        $j \leftarrow j + 1$ // Update the identifier of the fine-level layer.
16:    end while
17:    if $i > 1$ then
18:        Update the ∀-dominance between $L^{i-1}$ and $L^i$
19:    end if
20:    $\mathcal{L} \leftarrow \mathcal{L} \cup L^i$ // Insert $L^i$ into $\mathcal{L}$.
21:    $\mathcal{R} \leftarrow \mathcal{R} - L^l$ // Update target relation $\mathcal{R}$.
22:    $i \leftarrow i + 1$ // Update the identifier of the coarse-level layer.
23: end while
24: return $\mathcal{L}$
```

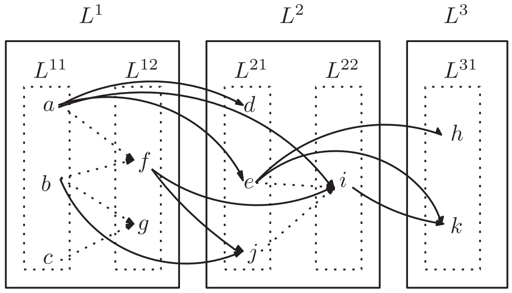


Fig. 5. Description of the dual-resolution layer.

3.2 Optimizing the Dual-Resolution Layer

In general, top-k queries have a relatively small retrieval size k compared to the cardinality of $\mathcal{R}$. From this observation, we identify some optimization room from the context of search engines, and devise *static indexing pruning* techniques [24] that facilitate to reduce index sizes. Specifically, when a *maximum retrieval size* k^{max} ($k \leq k^{max}$) is known beforehand, we develop two pruning techniques without compromising the accuracy of the top-k answers.

1) **Exploiting the number of fine-level layers:** No tuple in the ($k^{max}+1$)-th coarse-level layers can be in the top-k answers. Given the i-th coarse-level layer

L^i ($i \leq k^{max}$), we can also restrict the number of fine-level layers. Given the j-th fine-level layer L^{ij} in L^i, tuples in L^{ij} can be in a candidate set of the top-$(i+j-1)$ answers. If $i+j-1$ is greater than k^{max}, no tuples in L^{ij} can be in the top-k answers. Thus, we only construct L^{ij} such that $i+j-1 \leq k^{max}$ (lines 4 and 9 in Algorithm 1).

2) **Exploiting the number of dominance relationships:** Given a tuple t, if the number of $\forall$-dominance relationships for t is greater than or equal to k^{max}, t cannot be in the top-k answers. We can safely ignore t for top-k candidates. Before constructing the dual-resolution layer, these tuples can be eliminated using a k^{max}-*skyband* [25] that can be a tighter candidate set of the top-k answers than the k^{max} coarse-level layers. To further reduce the number of candidates, the optimization techniques proposed in the robust index [3] can be employed. We can estimate that such saving can be up to 20% from the ratio of the first fine-level layers (n = 1,000K, $d = 4$, $k^{max} = 10$ in independent distribution). Meanwhile, the building time of the reduced index is greater than that of the k^{max} skyline layers by up to five times (based on the evaluation reported in [6]).

We stress that the pruning techniques can also be applied for updating tuples. When new tuples are inserted or old tuples are deleted (to be explained in Section 7), we check if some tuples hold for either of two constraints for the number of fine-level layers and the number of dominance relationships. We can thus save the updating cost by safely pruning tuples outside k^{max} layers.

4 Top-*k* Query Processing

Top-k query processing over the layer-based index can be viewed as a *graph traversal problem* as discussed in the existing algorithms [4], [5]. Specifically, we form tuples as *nodes*, and $\forall$- and $\exists$-dominance relationships as *directed edges* between nodes. The key technique is to develop an efficient filtering method for avoiding unnecessary access. For this purpose, we fully exploit the $\forall$- and $\exists$-dominance relationships for the coarse- and fine-level layers, respectively.

First, $\forall$-dominance can be represented as *solid edges* (Fig. 5) between coarse-level layers L^i and L^{i+1} in a *bipartite graph*. The $\forall$-dominance relationship preserves the *monotonicity* in the coarse-level layer (Lemma 1). If $t \in L^i$ $\forall$-dominates $t' \in L^{i+1}$, a directed solid edge e is connected from t to t'. This can be used for selectively identifying the candidates of the top-k answers and avoiding the access to non-candidates.

Theorem 1 ($\forall$-dominance-ordered coarse layers). *Given coarse-level layers L^i and L^{i+1}, a tuple $t' \in L^{i+1}$ can be in the top-*k *answers if every tuple $t \in L^i$ $\forall$-dominating t' is in the top-$(k-1)$ answers.*

Second, $\exists$-dominance can be represented as *dotted edges* (Fig. 5) between adjacent fine-level layers L^{ij} and $L^{i(j+1)}$, preserving the *monotonicity* in the fine-level layer (Lemma 2). This can also be used for distinguishing the candidates of the top-k answers.

Algorithm 2 ComputeTopKProcessing($\mathcal{L}$, $\mathcal{F}$, k)

```
Input: $\mathcal{L}$: a dual layer, $\mathcal{F}$: a scoring function, $k$: a retrieval size
Output: $\mathcal{K}$: a set of top-$k$ tuples
1:  $\mathcal{K} \leftarrow \{\}$ // Initialize a top-$k$ answer set.
2:  $\mathcal{Q} \leftarrow \{\}$ // Initialize a priority queue for top-$k$ candidates.
3:  All tuples in $L^{11}$ are computed by $\mathcal{F}$, and inserted into $\mathcal{Q}$
4:  while $\mathcal{K}.size() \geq k$ do
5:      $t \leftarrow \mathcal{Q}.pop()$ // Pop $t$ with the smallest score from $\mathcal{Q}$.
6:      $\mathcal{K} \leftarrow \mathcal{K} \cup \{t\}$ // Insert $t$ into a top-$k$ answer set $\mathcal{K}$.
7:      for all $t'$ connected from $t$ in the coarse-level layer do
8:          if $t'$ is $\forall$-dominance-free and $\exists$–dominance-free then
9:              $t'$ is computed by $\mathcal{F}$ and is inserted into $\mathcal{Q}$
10:         end if
11:     end for
12:     for all $t'$ connected from $t$ in the fine-level layer do
13:         if $t'$ is $\forall$-dominance-free and $\exists$-dominance-free then
14:             $t'$ is computed by $\mathcal{F}$ and is inserted into $\mathcal{Q}$
15:         end if
16:     end for
17: end while
18: return $\mathcal{K}$
```

Theorem 2 ($\exists$-dominance-ordered fine layers). *Given fine-level layers L^{ij} and $L^{i(j+1)}$, a tuple $t' \in L^{i(j+1)}$ can be in the top-*k *answers if any tuple $t \in L^{ij}$ $\exists$-dominating t' is in the top-$(k-1)$ answers.*

Using these properties, we can maintain the *status* of a tuple t at each layer. Depending on the status, we can examine whether t needs to be accessed or not. Formally, the status of the tuple is defined as follows:

Definition 7 ($\forall$-dominance-free). *A tuple $t' \in L^{i+1}$ is $\forall$-dominance-free, if (1) t' has no connected edge from $t \in L^i$ or (2) every tuple $t \in L^i$ connected to t' is in the top-$(k-1)$ answers.*

Definition 8 ($\exists$-dominance-free). *A tuple $t' \in L^{i(j+1)}$ is $\exists$-dominance-free, if (1) t' has no connected edge from L^{ij} or (2) any tuple $t \in L^{ij}$ connected to t' is in the top-$(k-1)$ answers.*

Example 3 (Status of tuples). *We continue the example for the dual-resolution layer (Fig. 5). We can identify that tuples $\{a, b, c, f, g\}$ in the first coarse-level layer are $\forall$-dominance-free. In addition, tuples $\{a, b, c, d, e, j, h, k\}$ in the first fine-level layer of each coarse-level layer are $\exists$-dominance-free. When sequentially accessing tuples over the dual-resolution layer, the statuses of tuples can be changed. If a and f in L^1 are in the top-$(k-1)$ answers, i is changed to be $\forall$-dominance-free. Similarly, if a or b in L^{11} is added to the top-$(k-1)$ answers, f is changed to be $\exists$-dominance-free.*

On the basis of Theorems 1 and 2, when traversing tuples through the dual-resolution layer, we only compute tuples that are both $\forall$-dominance-free and $\exists$-dominance-free, and avoid the unnecessary access to non-candidates. Formally:

Theorem 3 (Filtering condition). *A tuple t has to be only accessed if t is both $\forall$-dominance-free and $\exists$-dominance-free.*

We explain the overall procedure for computing top-k processing over the dual-resolution layer (Algorithm 2). Let $\mathcal{Q}$ be a priority queue in which tuples are placed in increasing order of scores by $\mathcal{F}$. First, we insert all tuples in L^{11} into $\mathcal{Q}$, because they are both $\forall$-dominance-free and $\exists$-dominance-free (line 3). We then pop a tuple t with

TABLE 1
Top-k Query Processing over the Dual-Resolution Layer

Step	Action	$\mathcal{Q}$	$\mathcal{K}$
1	Initialize $\mathcal{Q}$ and $\mathcal{K}$.	$\{\}$	$\{\}$
2	Access tuples in L^{11}.	$\{a,b,c\}$	$\{\}$
3	Pop a from $\mathcal{Q}$.	$\{b,c\}$	$\{a\}$
4	Update $\{d,e,f,i\}$ connected to a.	$\{b,f,d,e,c\}$	$\{a\}$
5	Pop b from $\mathcal{Q}$	$\{f,d,e,c\}$	$\{a,b\}$
6	Update $\{g,j\}$ connected to b.	$\{f,d,e,c,g\}$	$\{a,b\}$
7	Pop f from $\mathcal{Q}$	$\{d,e,c,g\}$	$\{a,b,f\}$

the smallest score from $\mathcal{Q}$ and insert t into a final top-k answer set (lines 5–6). We also check the filtering condition (Theorem 3) for other tuples connected to t in coarse- and fine-level layers (lines 7–16). This process is iterated until the top-k answers are identified.

Example 4 (Top-k query processing). *Suppose that a retrieval size k is 3, and a weight vector w is (0.5, 0.5). Table 1 describes the action of our proposed top-k query processing. After initializing a priority queue $\mathcal{Q}$ and an answer set $\mathcal{K}$, we access tuples $\{a,b,c\}$ in L^{11} and enqueue them into $\mathcal{Q}$. Because tuple a is the top-1 tuple, we pop a from $\mathcal{Q}$. We then update the status of tuples $\{d,e,f,i\}$ connected to a. Because tuples $\{d,e,f\}$ are changed to be $\forall$-dominance-free and $\exists$-dominance-free, we access those tuples and insert them into $\mathcal{Q}$. As the top-2 tuple, we pop b from $\mathcal{Q}$ and also update the status of tuples $\{g,j\}$ connected to b. Because g becomes $\forall$-dominance-free and $\exists$-dominance-free, g is inserted into $\mathcal{Q}$. Lastly, we pop f from $\mathcal{Q}$ as the top-3 tuple. As a result, the top-3 answers are $\{a,b,f\}$.*

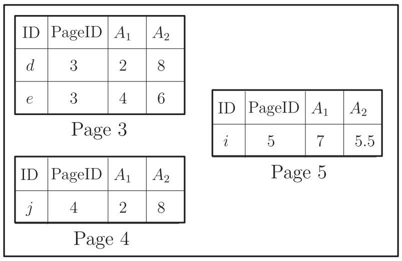
Lastly, we discuss the efficiency of top-k query processing by analytically proving that our top-k query processing incurs *less* cost than DG [4], [6]. (DG is known as the most efficient layer-based index.) Specifically, DG can be viewed as a layer-based index that employs only coarse-level layers from dual-resolution layer indexing and cannot take advantage of $\exists$-dominance relationships. We define the *cost* used for evaluating our top-k query processing, and analyze the cost of our top-k query processing. Note that the cost model is consistently used in DG.

Definition 9 (Cost). *The cost is the number of tuples that are both accessed and computed by $\mathcal{F}$ during top-k query processing.*

Theorem 4 (Cost analysis). *Our top-k query processing over the dual-resolution layer always incurs less or equal cost than DG.*

5 Disk-Based Dual-Resolution Layer

In this section, we discuss a disk-based scheme for the dual-resolution layer. In general, it is impractical to assume that all tuples and indexing structures are loaded into main memory. When top-k queries are executed, I/O cost can be another key factor. We formally define I/O cost as *the number of I/O accesses* during top-k query processing. Note that the disk-based scheme only minimizes the number of I/O cost during top-k query processing, and it does not affect the number of tuples accessed.



ID	PageID	A_1	A_2
d	3	2	8
e	3	4	6

Page 3

ID	PageID	A_1	A_2
j	4	2	8

Page 4

ID	PageID	A_1	A_2
i	5	7	5.5

Page 5

Fig. 6. Page arrangement for L^2 in a data file.

Basically, the disk-based layer is partitioned by two files, an *index file* and a *data file* that store the dual-resolution layer and original tuple values, respectively. This is similar to the disk-based DG [6]. For each tuple in the dual-resolution layer, the index file stores $\langle$*tuple ID*, *page ID*, *relationships*$\rangle$ in a hash table. The data file stores multiple tuples in a default page size (*e.g.*, 4KB) as an array data file, where a tuple consists of $\langle$*tuple ID*, *attribute values*$\rangle$. All tuples in L^{ij} are thus stored in $\lceil \frac{|L^{ij}| \times tuple\_size}{page\_size} \rceil$ pages, where *tuple_size* is the byte size per tuple and $|L^{ij}|$ is the number of tuples in L^{ij}.

We now discuss how to build a data file to reduce the I/O cost. Toward this goal, we need to consider (1) how to measure the probability of two tuples being accessed together, and (2) how to group m tuples in the same layer. To resolve these problems, we assume that if two tuples t and t' share similar *parent sets*, they are highly likely to be accessed together. Given $t \in L^i$, a coarse-level parent set $CPar(t)$ is defined by a set of tuples in L^{i-1} that $\forall$-dominate t. Similarly, given $t \in L^{ij}$, a fine-level parent set $FPar(t)$ is defined by a set of tuples in $L^{i(j-1)}$ that $\exists$-dominate t. For instance, given the dual-resolution layer in Fig. 5, $CPar(t)$ and $FPar(t)$ of tuple i in L^{22} are $\{a,f\}$ and $\{e,j\}$, respectively.

We then measure the distance between tuples using coarse- and fine-level parents. Specifically, we adopt *hamming distance* for calculating the distance between tuples as used in [6]. Given two tuples t and t', coarse-level distance is defined as $CDist(t,t') = |CPar(t) \cup CPar(t')| - |CPar(t) \cap CPar(t')|$. Similarly, fine-level distance is defined as $FDist(t,t') = |FPar(t) \cup FPar(t')| - |FPar(t) \cap FPar(t')|$. The smaller the distance is, the higher the probability of two tuples being accessed together is.

After coarse- and fine-level distances of tuple pairs are computed, tuples are grouped using a *hierarchical clustering method*. Specifically, a pair of tuples with the smallest coarse-level distance is chosen and fine-level distance is used as a tie-breaker for coarse-level distance. After the pair of tuples is grouped, we need to recompute coarse- and fine-level parent sets for a grouped tuple set G. By unifying the parent sets for the grouped tuples, $CPar(G)$ is updated by $\{CPar(t)|t \in G\}$. Similarly, $FPar(G)$ is updated by $\{FPar(t)|t \in G\}$. In addition, coarse- and fine-level distances of G are updated. This procedure iteratively continues in a bottom-up fashion until the size of grouped tuple sets is approximate to (but no more than) $\lfloor \frac{page\_size}{tuple\_size} \rfloor$.

To illustrate this, Fig. 6 shows how tuples in L^2 are stored in a data file. Suppose that a disk page stores two tuples. Given L^{21}, because d and e share the same coarse-level parent sets (Fig. 5), they are allocated in the same page. In addition, $|L^{22}|$ is smaller than the page size, it is simply stored in one page. Using this data-based scheme, we can reduce I/O cost by accessing the multiple tuples sharing similar relationships.

6 Optimized Top-*k* Query Processing

This section presents optimization techniques for reducing the access of top-1 candidates in L^{11}. Because all tuples in L^{11} are ∀- and ∃-dominance-free, we have to access all tuples in L^{11} (line 3 in Algorithm 2). To alleviate this problem, we propose the optimization of forming a *virtual layer* L^0, which guides selective access for top-1 candidates in L^{11}. (Section 8.4 shows that this optimization reduces the number of accessed tuples compared to our previous heuristic method [1]).

For simplicity, we first explain how to build virtual layer L^0 in two-dimensional space. Ideally, given a weight vector $w = (w_1, w_2)$, only one tuple in L^{11} can be identified for the top-1 answer. Let $\mathcal{W}$ denote a set of all possible weight vectors w. In the two-dimensional case, the following conditions hold: $0 < w_1, w_2 < 1$ and $w_1 + w_2 = 1$. Using the condition between w_1 and w_2, $\mathcal{W}$ can be represented by one-dimensional *ranges* of w_1, and the corresponding ranges of w_2 can be determined by $1 - w_1$. $\mathcal{W}$ consists of a finite set of disjoint ranges of w_1, *i.e.*, $\{W^1, \ldots, W^{|L^{11}|}\}$ such that $\bigcup_{i=1}^{|L^{11}|} W^i = [0{:}1]$ and $W^i \cap W^j = \emptyset$ $(i \neq j)$. When each range $W \in \mathcal{W}$ is associated with one tuple, it is effective for identifying exact the top-1 answer. However, it is non-trivial for extending it for three or higher dimensional data.

As an alternative way, we assign a set of c candidates for partitioned vectors representing ranges. Note that c is always greater than or equal to dimensionality d, *i.e.*, $c \geq d$. Initially, given two vectors (1, 0) and (0, 1) in two-dimensional space, we compute c candidate tuples (c = 2) and check if they share one *common* tuple. When they do not share one tuple, we recursively partition the range represented by two vectors, *i.e.*, 90 degree angle between two vectors (1, 0) and (0, 1) is split into equal 45 degrees using (0.5, 0.5) as boundary. This process continues until adjacent vectors always share one common tuple. (When knowing the distribution of tuples in prior, this equal-degree recursive partitioning can be performed more efficiently as discussed in angle-based partitioning [26].) In this case, a set of top-1 candidates is the union of top-c candidates for the two vectors. Formally:

Theorem 5 (Top-1 candidates). *Given weight vector w in d-dimensional space, if the range represented by d vectors covers w and each adjacent pair of vectors shares one common tuple, the union of top-c tuples ($c \geq d$) for d vectors includes the top-1 answer for w.*

Proof. We prove this by contradiction. Given a weight vector w, assume that top-1 tuple t is not contained in the union of top-c tuples for d adjacent vectors of w. Given w, the score $\mathcal{F}_w(t)$ of t has to be smaller than the score $\mathcal{F}_w(t^s)$ of a common tuple t^s for d weight vectors. Because w and d weight vectors preserve monotonicity, $\mathcal{F}_w(t)$ cannot be smaller than $\mathcal{F}_w(t^s)$ for every weight vector, which is a contradiction. □

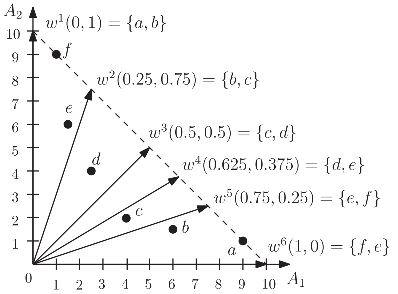


Fig. 7. Weight ranges for top-1 candidate tuples (c = 2).

Example 5 (First layer optimization). *Fig. 7 depicts all possible ranges in which each vector is associated with top-2 tuples ($c = 2$). Two adjacent vectors represent ranges. Consider that the first layer L^{11} consists of six tuples $\{a, b, c, d, e, f\}$. Two vectors $w^1 = (0, 1)$ and $w^6 = (1, 0)$ are used by initial vectors, where top-2 tuples are $\{a, b\}$ and $\{f, e\}$, respectively. As they do not share one tuple, we partition the region between w^1 and w^6, and identify $w^3 = (0.5, 0.5)$ in which top-2 tuples are $\{c, d\}$. After partitioning the range between w^1 and w^3, $w^2 = (0.25, 0.75)$ with $\{b, c\}$ is identified. In this case, w^1 and w^2 share b, and w^2 and w^3 share c. Similarly, after partitioning the range between w^3 and w^6, $w^4 = (0.625, 0.375)$ and $w^5 = (0.75, 0.25)$ are further obtained, where $\{d, e\}$ and $\{e, f\}$ are the top-2 tuples. Finally, this recursive partitioning guarantees that every adjacent vector pair shares one common tuple.*

This recursive partitioning can be extended for three or higher dimensional space. When $d = 3$, consider three vectors (1, 0, 0), (0, 1, 0), (0, 0, 1) in an initial case. By projecting three-dimensional space into two-dimensional space, we first partition (1, 0, 0) and (0, 1, 0), where the third dimension is fixed. After that, each range is further partitioned for (0, 0, 1), and a set of top-1 candidates is identified from three adjacent vectors. Therefore, the size of top-1 candidate set is at most dc, implying the size of the union of top-c tuples for d vectors.

To find a set of top-1 candidates, d vectors are identified as the range covering a given weight vector. We use the union of top-c tuples for d vectors as top-1 candidates. For instance, given $w = (0.3, 0.7)$, w^2 and w^3 are two vectors encompassing w, and $\{b, c, d\}$ is a set of top-1 candidates. As the proposed partitioning can preserve the *proximity* between adjacent ranges, ∃-dominance relationships can be built between tuples in adjacent vectors. Using the ∃-dominance relationships, when performing top-k query processing, we can selectively access tuples for the top-k answers, and reduce the number of accessed tuples than clustering-based partitioning [1], [4], [6].

7 Index Maintenance

In this section, we present efficient maintenance algorithms in dynamic data environments. For insertion and deletion operations, the dual-resolution layer is maintained in an incremental manner.

7.1 Insertion Operation

We develop the insertion algorithm of the dual-resolution layer (Algorithm 3). When a new tuple t^{new} comes in, we

Algorithm 3 Insertion(t^{new}, $\mathcal{L}$)

```
Input: $t^{new}$: an inserted tuple, $\mathcal{L}$: a target dual layer
Output: $\mathcal{L}$: an updated dual layer
 1: $i \leftarrow 1$ // Initialize a coarse-level id of $\mathcal{L}$.
 2: while $i \leq l$ do
 3:     if $\nexists t \in L^i : t \prec t^{new}$ then
 4:         Break
 5:     else
 6:         $i \leftarrow i + 1$ // Move to the next layer.
 7:     end if
 8: end while
 9: Insert $t^{new}$ into $L^i$
10: Set a fine-level id of $t^{new}$ as 1 // Update fine-level layer $L^{i1}$.
11: $S \leftarrow \{t' \in L^i | t^{new} \prec t'\}$ // Initialize updated tuples in $L^i$.
12: for $j \leftarrow i$ to $l$ do
13:     if $j > i$ then
14:         $S' \leftarrow \{t' \in L^j | \exists t \in S : t \prec t'\}$ // Find updated set $S'$.
15:         $S \leftarrow S'$ // $S$ is updated for $S'$ in $L^j$.
16:     end if
17:     for $\forall t \in S$ do
18:         UpdateFineLevelLayer($t$, $L^j$)
19:         $t$ is degraded from $L^j$ to $L^{(j+1)}$
20:         Set a fine-level id of $t$ as 1 // Update $L^{(j+1)1}$.
21:         Build the $\forall$-dominance between $L^j$ and $t$
22:     end for
23: end for
24: Build the $\forall$-dominance between $L^{i-1}$ and $t^{new}$
25: Build the $\forall$-dominance between $L^{i+1}$ and $t^{new}$
26: return $\mathcal{L}$
```

Algorithm 4 Deletion(t^{old}, $\mathcal{L}$)

```
Input: $t^{old}$: a deleted tuple, $\mathcal{L}$: a target dual layer
Output: $\mathcal{L}$: an updated dual layer
 1: Find a coarse-level layer $L^i$ including $t^{old}$
 2: UpdateFineLevelLayer($t^{old}$, $L^i$)
 3: Erase $\forall$-dominance between $L^{(i-1)}$ and $t^{old}$
 4: Erase $t^{old}$ from $L^i$
 5: $S \leftarrow \{t \in L^{i+1} | t^{old} \prec t, \nexists t' \in L^i\} : t' \prec t\}$
 6: for $j \leftarrow i + 1$ to $l$ do
 7:     if $j > i + 1$ then
 8:         $S' \leftarrow \{t \in L^j | \exists t' \in S : t' \prec t, \nexists t'' \in L^{j-1} : t'' \prec t\}$
 9:         $S \leftarrow S'$
10:     end if
11:     for $\forall t \in S$ do
12:         UpdateFineLevelLayer($t$, $L^j$)
13:         $t$ is upgraded from $L^j$ to $L^{j-1}$
14:         Set a fine-level id of $t$ as 1
15:         Build $\forall$-dominance between $L^{(j-2)}$ and $t$
16:     end for
17: end for
18: return $\mathcal{L}$
```

first find the insertion location of t^{new} in coarse-level layers by performing dominance tests between $t \in L^i$ and t^{new}. If t^{new} is not dominated by any other tuples in L^i, the location of t^{new} becomes L^i. Otherwise, the tuples in the next layer are compared with t^{new} (line 2–8). After that, t^{new} is inserted into L^i (line 9). Given a coarse-level layer L^i, we then update fine-level layers within L^i. Because the updating cost for fine-level layers is known to be expensive [2], [3], we adopt a *lazy* method to update fine-level layers. By simply setting the fine-level identifier of t^{new} as 1, t^{new} is inserted into L^{i1} (line 10). This lazy method holds for the correctness of top-k query processing, as it does not affect the existing fine-level relationships. In addition, because t^{new} can affect tuples in the inserted layer and all the other layers $\{L^j \in \mathcal{L} | i \leq j \leq l\}$ (line 12), some tuples are moved from L^j to L^{j+1}. Let S be a set of updated tuples that are degraded to the next layer. For each coarse-level layer, S is iteratively identified (lines 14-15) and both $\forall$- and $\exists$-dominance relationships of S are updated. The existing $\exists$-dominance relationships of S are first erased (line 18). For new $\exists$-dominance relationships, the updated tuples are inserted into the first fine-level layer as done in t^{new} (line 20). In this process, to further optimize update cost, we can remove some updated tuples by checking if pruning conditions for the number of fine-level layers and the number of dominance relationships hold (as discussed in Section 3.2).

Although the lazy method can significantly reduce the computation cost for updating fine-level layers, the cost of top-k query processing becomes worse, implying that most of the tuples are located in the first fine-level layers. In the worst case, the cost of the lazy method converges to that of DG. When performing the lazy method, the dual-resolution layer needs to be rebuilt periodically. As the opposite case, we can rebuild the layer for each inserted tuple, called an *eager* method.

As a moderate method, we devise a *lightweight* alternative by combining the lazy and eager methods. The key intuition is that we calculate the ratio of the number of appended tuples in the first fine-level layers. As tuples are moved to the first fine-level layer, the ratio increases. When the ratio exceeds a threshold, we perform rebuilding the dual-resolution layer as in the eager method. Otherwise, we only update the fine-level ids of tuples as in the lazy method. (In Section 8.9, we empirically compare the three methods).

Lastly, we discuss how to update tuples in the disk-based storage scheme. When some tuples are inserted or deleted, the locations of tuples are changed over the layer. In this case, we also need to recompute the locations of tuples for maintaining the proximity of tuples in the disk-based scheme. For updated layers, we perform disk-based operations such as *insertion*, *deletion*, *split*, and *merge* as discussed in [6]. When the updated tuples do not affect splitting or merging the existing page, we only recompute a specific disk page. Otherwise, we need to merge two or more pages into one page. In addition, when the size of tuples exceeds the page size, we have to split one page into two pages. For merge and split operations, we first collect tuples from updated pages, and then perform a hierarchical clustering for maintaining the proximity of tuples as building the disk-based scheme. (Due to the space limitation, we leave the details of the disk-based operations, which are similar to [6]).

7.2 Deletion Operation

The simplest way to carry out deletion operations is to mark deleted tuples as *pseudo-tuples* that are virtual tuples used only for top-k query processing. Although the maintenance cost of this method is efficient, the access cost of top-k query processing increases rapidly with the number of tuples deleted.

We develop an alternative deletion algorithm over the dual-resolution layer (Algorithm 4). When deleting an old

TABLE 2
Index Construction Time (Sec) of Algorithms
($k = 10$, $d = 4$, $n = 1000K$)

Dist.	HL	HL+	DG	DG+	DL	DL+
IND	4.24	4.24	12.828	12.867	24.195	24.261
ANT	42.462	42.462	469.538	484.46	608.39	611.019

TABLE 3
Construction Time of DL+ (Sec) in Anti-Correlated Distribution
($k^{max} = 30$, $n = 200K$)

Method	$d = 2$	$d = 3$	$d = 4$	$d = 5$
w/o pruning	7.388	49.776	237.819	812.486
w/ pruning	2.269	5.030	29.242	264.651

tuple t^{old}, we traverse coarse-level layers to identify the location of t^{old} in breadth-first search. After that, the ∀-dominance relationships of t^{old} are erased in L^{i-1}, the ∃-dominance relationships of t^{old} are updated by calling the **UpdateFineLevelLayer** function as lazy updates (lines 2–3), and t^{old} is deleted from L^i (line 4). Similar to the insertion operation, t^{old} can affect tuples in all the other layers $\{L^j \in \mathcal{L} | i+1 \leq j \leq l\}$ (line 6). We thus check if some tuples in L^j are only dominated by t^{old} (lines 7–10). Once a set S of updated tuples are found, both existing ∀- and ∃-dominance relationships of S are erased and they are moved from L^j to L^{j-1} (lines 12–13). In addition, the fine-level identifer of S is set as 1 (line 14). Lastly, new ∀-dominance relationships for S are built for L^{j-2} (line 15). As discussed in insertion operations, we can also perform three methods for deletion operations, *i.e.*, eager, lazy, and hybrid methods. (We compare the three methods for deletion operations in Section 8.9).

8 Experiments

8.1 Experimental Settings

To evaluate our proposed dual-resolution layer, we generated extensive synthetic datasets with four parameters: distribution, dimensionality, retrieval size, and cardinality. All the attribute values were positive real numbers.

- Distribution: We generated two datasets, Independent (IND) and Anti-correlated (ANT), following the data generation instructions in [19]
- Dimensionality d: We varied the dimensionality d from 2 to 5. (Default: $d = 4$)
- Cardinality n: We varied the cardinality n from 200K to 1,000K. (Default: $n = 1,000K$)
- Retrieval size k: We varied the retrieval size k from 10 to 100. (Default: $k = 30$)

For scoring function $\mathcal{F}$, we randomly generated a user-specific weight vector $w = (w_1, \ldots, w_d)$ satisfying $\forall i \in [1, d]$: $0 < w_i < 1$ and $\sum w_i = 1$.

We then compared our dual-resolution layer with the state-of-the-art layer-based algorithms.

- DG+: This is the advanced algorithm of the *dominant graph* (DG) [4] which finds skylines iteratively and materializes them as layers. The dominance relationships between layers enable selective access to layers. We built a zero layer L^0 for the first layer by exploiting the pseudo-tuples using k-means clustering. For this optimization technique, we followed the instructions explained in [4]. To compute the skylines, we employed the state-of-the-art skyline algorithm BSkyTree [27].
- HL+: This is the optimized version of the *hybrid layer* (HL) [5] which materializes convex skyline as layers, but builds d sorted lists within each layer for d attributes. When performing TA [10] using sorted lists for each layer, we updated a tighter threshold by accessing convex layers in a round-robin manner as described in [5]. To compute convex skylines, we modified the state-of-the-art convex hull algorithm QHull [28] available at *http://www.qhull.org*.
- DL+: We optimized our proposed dual-resolution layer (DL) by building a zero layer L^0 as discussed in Section 6. To compute skylines and convex skylines for the dual-resolution layer, we employed the same algorithms used in DG+ and HL+.

To validate the performance, we employed *the number of tuples evaluated* (Definition 9) and *the number of I/O accessed*. Another measure, *query response time* may affect detailed implementation techniques of each indexing methods, and the response time tends to be proportional to our measures. Therefore, our measures are suitable for comparing the overall performance.

All experiments were conducted using Windows 7 running on an Intel Core i7 950 3.07 GHz CPU with 24GB RAM. All algorithms are implemented by C++.

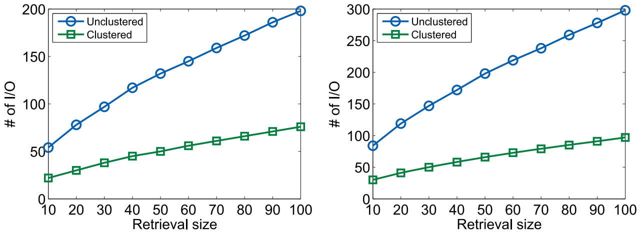
8.2 Index Construction Time

We first compared the index construction time of algorithms (Table 2). Each parameter was set by default for each distribution.

The dominant part of the index building time comes from making the layers and forming the relationships between layers. Specifically, DG computes skylines for each layer and connects the relationship between adjacent layers. The index building time of HL includes the convex skyline computation for each layer and the sorting of attribute values within each layer. In contrast, since DL requires the computation of both skylines and convex skylines for coarse- and fine-level layers, it encodes richer relationships between tuples but incurs more computation time than other indices.

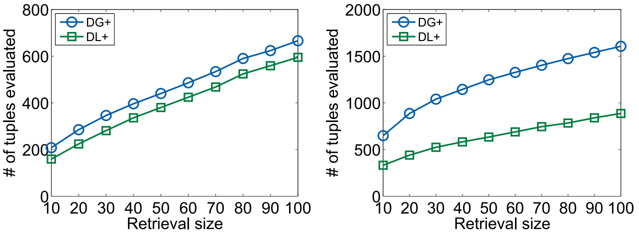
In addition, to optimize zero layer L^0 in DG and DL, the optimized versions, DG+ and DL+, require more construction time. Because the number of tuples in the zero layer is small, however, this computation time is negligible. Meanwhile, HL+ shares the same index with HL (*i.e.*, the different part is only the query processing using a tighter threshold), and thus incurs the same index building time.

We then report on the construction time for the dual-resolution layer using two pruning techniques. Suppose that maximum retrieval size k^{max} is 30 and n = 200K. To filter out unnecessary tuples, we first compute k^{max}-skyband, and then sequentially build each layer L^{ij} satisfying the condition $i + j - 1 \leq k^{max}$. Table 3 illustrates the construction time for anti-correlated distribution over varying



(a) (b)

Fig. 8. I/O cost for top-k query processing. (a) IND. (b) ANT.



(a) (b)

Fig. 10. DG+ and DL+ with varying retrieval size k. (a) IND. (b) ANT.

dimensionality. It is clear that the dual-resolution layer using pruning techniques significantly reduces the construction time by 3-8 times. In addition, since the optimized version of DL only keeps the number of relationships for each tuple that is less than k^{max}, it can skip checking the relationships of unnecessary tuples during top-k query processing.

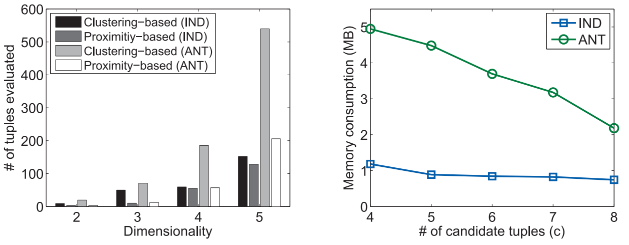
8.3 Optimization for I/O Cost

This section validates our proposed data storage scheme to minimize I/O cost. Each parameter was set $n = 200K$ and $d = 4$. Assume that each disk page stores 10 tuples. If a tuple is accessed from the disk file, other tuples in the same page are accessed together, and they are cached in main memory. Suppose that the cached tuples are kept during top-k query processing.

We compared two disk-based storage schemes: (1) Unclustered collects tuples in a sequential manner, and (2) Clustered stores tuples using a hierarchical clustering method, where the distance between tuples is defined as hamming distance (as discussed in Section 5). Fig. 8 shows I/O cost in different distributions over varying retrieval sizes. It is clear that Clustered reduces the number of I/O than Unclustered in all parameter settings. In addition, because the number of I/O is less than 100 pages, memory consumption of Clustered is less than 1 MB, where the size of one page is 1 KB.

8.4 Optimization for the First Layer

This section compares our proximity-based method for the first layer with the existing clustering-based method [1]. For each method, we measure the number of tuples evaluated while identifying the top-1 answer. We used independent and anti-correlated datasets with d from 2 to 5 (n = 1,000K, $c = 4$). We also measure memory consumption with c from 4 to 8 (n = 1,000K, d = 4).



(a) (b)

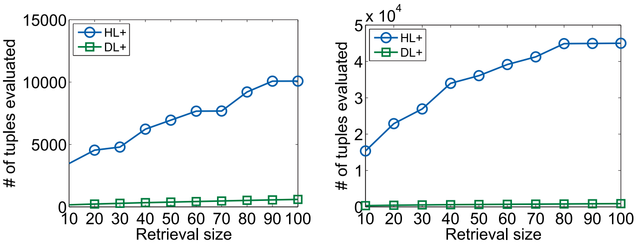
Fig. 9. Number of accessed tuples and memory consumption for the top-1 answer. (a) # of tuples evaluated. (b) Memory consumption.

Fig. 9(a) compares the pruning effect using two optimization methods. As the dimensionality increases, the number of evaluated tuples in clustering-based method increases exponentially. In contrast, the number of evaluated tuples in the proximity-based method increases linearly. When $d = 5$, the performance gap is up to 3 times in anti-correlated datasets. In addition, Fig. 9(b) reports on memory consumption over varying c. As c increases, the memory consumption tends to decrease. This is because the number of vectors used for representing possible ranges decreases. Meanwhile, the number of tuples evaluated increases conversely with c. As we mainly focus on minimizing the number of tuples evaluated, we set $c = d$ by default in our evaluation.

8.5 Effect of Retrieval Size

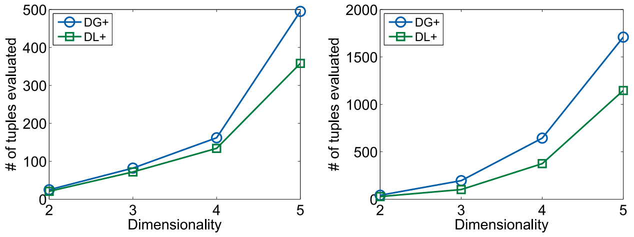
This section compare three algorithms over varying retrieval sizes. We used independent and anti-correlated datasets with k from 10 to 50 ($d = 4$, $n = 200K$). In all settings, DL+ always outperforms DG+ and HL+. Specifically, regardless of the change in the retrieval sizes, DL+ consistently outperforms DG+ (Fig. 10). For instance, when $k = 50$, DL+ consistently accesses about two times fewer tuples than DG+ in anti-correlated distribution, regardless of retrieval sizes.

We next compare DL+ and HL+ over varying retrieval sizes. Observe that DL+ accesses far fewer tuples than HL+. In particular, as the retrieval size increases, the performance gap between DL+ and HL+ tends to increase (Fig. 11). For instance, when $k = 50$ in anti-correlated distribution, the access cost of DL+ is one order of magnitude smaller than that of HL+. This is because the selective access in HL+, performing TA for sorted lists, is sensitive to retrieval sizes.



(a) (b)

Fig. 11. HL+ and DL+ with varying retrieval size k. (a) IND. (b) ANT.



(a) (b)

Fig. 12. DG+ and DL+ with varying dimensionality *d*. (a) IND. (b) ANT.

8.6 Effect of Dimensionality

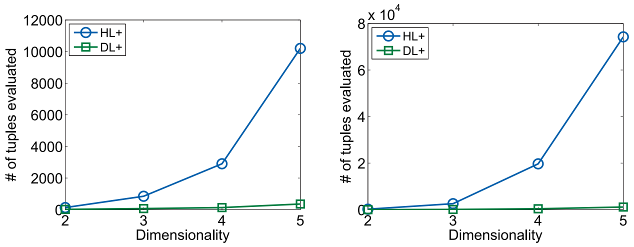
This section evaluates the effect of dimensionality by comparing three algorithms. We used independent and anti-correlated datasets with d from 2 to 5 (k = 10, n = 200K). In all settings, DL+ always accesses fewer tuples than DG+ and HL+.

As the dimensionality increases, the performance gap between DL+ and DG+ increases linearly (Fig. 12). When $d = 5$, DL+ accesses about 1.5 times fewer tuples than DG+ in anti-correlated distribution. This observation suggests that the optimization margin coming from the filtering technique using fine-level layers increases in high-dimensional data. Specifically, since our key contribution is the division of coarse-level layers and the encoding of richer dominance relationships, *i.e.*, ∀- and ∃-dominance, DL+ is relatively more efficient. In contrast, other methods suffer from accessing coarse-level layers with too many tuples. It is known from the previous literature that such cardinality explodes when the dimensionality is high or attributes are anti-correlated, which is known as the curse of dimensionality problem. In this sense, using ∃-dominance relationships is more effective for high dimensional and anti-correlated data.

We next compare DL+ with HL+ for varying dimensionality (Fig. 13). Observe that DL+ accesses far fewer tuples than HL+. When $d = 5$ in anti-correlated distribution, the access cost of DL+ is two orders of magnitude smaller than that of HL+.

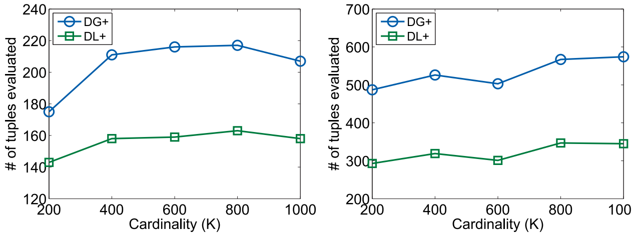
8.7 Effect of Cardinality

This section evaluates the effect of cardinality by comparing two algorithms. We used independent and anti-correlated datasets with n from 100K to 500K (k = 10, d = 4). In Fig. 14, we found that DL+ always outperforms DG+. In addition, it shows that both DL+ and DG+ are less sensitive to cardinality n. This is because the number of tuples within each layer does not increase, but the number of layers increases with cardinality. When $k \ll n$, DL+ and DG+ are thus insensitive to n.



(a) (b)

Fig. 13. HL+ and DL+ with varying dimensionality *d*. (a) IND. (b) ANT.



(a) (b)

Fig. 14. DG+ and DL+ with varying cardinality *n*. (a) IND. (b) ANT.

8.8 Evaluation on Real-Life Datasets

In this section, we examine the proposed dual-resolution layer using two real-life datasets such as NBA and House, which are collected from *http://www.nba.com* and *http://www.ipums.org* respectively. They consist of 17,264 8-dimensional tuples and 127,931 6-dimensional tuples. Compared to synthetic datasets, real-life datasets follow correlated and independent distribution, respectively.

We report on the number of accessed tuples of algorithms over varying retrieval sizes (Table 4). The performance gap narrows down, because they were at a smaller scale than synthetic datasets. However, DL+ consistently outperformed HL+ and DG+ in both real-life datasets as shown in synthetic datasets. As the retrieval size increases, the gap between DL+ and DG+ increases as well. For instance, when k = 90, DL+ saves 10% more than DG+ in both real-life datasets.

8.9 Evaluation for DL Maintenance

In this section, we evaluate our proposed maintenance algorithms for the dual-resolution layer. For insertion operations, we randomly generated new 10K tuples in independent distribution ($n = 200K, d = 4, k = 10$). For deletion operations, we randomly chose 10K tuples from the existing datasets. Assume that inserted or deleted tuples are processed one by one in a sequential manner. The maintenance time is computed by the sum of the given insertion or deletion operations. After performing the maintenance algorithms, the average number of tuples evaluated was measured for 100 random queries.

TABLE 4
Comparisons with State-of-the-Art Algorithms over Varying Retrieval Sizes

Algorithm	10	30	50	70	90
HL+ (NBA)	716	3,135	4,849	5,829	6,188
DG+ (NBA)	167	399	689	1,014	1,283
DL+ (NBA)	165	388	626	897	1,100
HL+ (House)	12,145	13,804	14,721	14,916	15,778
DG+ (House)	1,211	1,574	1,899	2,173	2,415
DL+ (House)	1,010	1,345	1,676	1,965	2,191

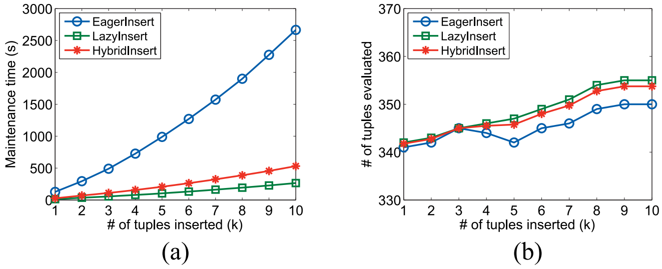


Fig. 15. Comparisons of three insertion methods. (a) Maintenance time. (b) # of tuples evaluated.

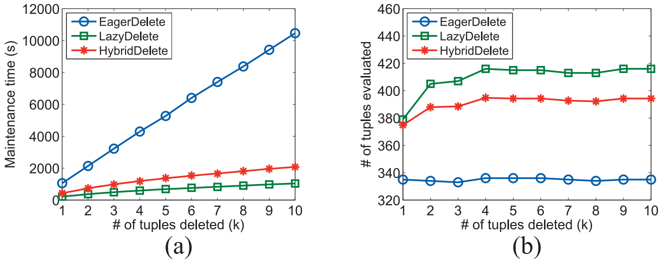


Fig. 16. Comparisons of three deletion methods. (a) Maintenance time. (b) # of tuples evaluated.

8.9.1 Insertion Operations

To validate our proposed maintenance algorithm for insertion operations, we implemented the following three methods:

- EagerInsert: This maintains coarse-level layers in an incremental fashion, and reconstructs all fine-level layers for updated coarse-level layers.
- LazyInsert: Algorithm 3 describes the overall procedure of LazyInsert.
- HybridInsert: This combines the eager and lazy method by using a threshold for the ratio of the number of appended tuples in the first fine-level layer. Empirically, we set the threshold as 0.1. When the number of appended tuples exceeds $0.1 \times \sum_{i=1}^{l} |L^{i1}|$, we rebuild the layer as in EagerInsert. Otherwise, we update the fine-level ids of tuples as in LazyInsert.

We compared the three methods for insertion operations (Fig. 15). For the maintenance time, LazyInsert is significantly faster than EagerInsert (Fig. 15(a)). It is clear that the gap between EagerInsert and LazyInsert increases exponentially with the number of inserted tuples. This is because the computation cost for updating fine-level layers increases exponentially. In addition, HybridInsert shows the moderate performance between EagerInsert and LazyInsert. For the number of tuples evaluated, LazyInsert is more costly than EagerInsert (Fig. 15(b)). Meanwhile, HybridInsert consistently outperforms LazyInsert, which implies it rebuilds the layer less conservatively than LazyInsert. In this sense, HybridInsert can be considered as a balanced medium of EagerInsert and LazyInsert.

8.9.2 Deletion Operations

We implemented the following two algorithms for deletion operations:

- EagerDelete: Similar to EagerInsert, this consecutively maintains coarse-level layers, and reconstructs all fine-level layers for updated coarse-level layers.
- LazyDelete: Algorithm 4 describes the overall procedure of LazyDelete.
- HybridDelete: Similar to HybridInsert, we empirically set the threshold as 0.1. When the number of appended tuples exceeds $0.1 \times \sum_{i=1}^{l} |L^{i1}|$, we rebuild the layer as in EagerDelete. Otherwise, we only update fine-level ids of tuples as in LazyDelete.

We then validated the efficiency of LazyDelete (Fig. 16). Similar to the insertion operations, LazyDelete is much faster than EagerDelete in terms of maintenance overhead (Fig. 16(a)). In particular, when $10K$ tuples are deleted, the gap between EagerDelete and LazyDelete is up to one order of magnitude. For the number of tuple evaluated, LazyDelete is more costly than LazyInsert. In addition, HybridDelete shows the moderate performance between EagerDelete and LazyDelete as shown in Fig. 16(a) and (b).

9 Related Work

Top-k queries are widely used in many real applications such as image [8], [9] and text [13] retrieval. The main concern for top-k query processing is to reduce the number of tuples accessed from the target relation as much as possible. A survey of top-k query processing can be found in [29].

9.1 Layer-Based Approach

The layer-based approach materializes tuples into consecutive layers, and employs the relationships between layers to reduce data access. We categorize the existing layer-based approaches into three types: skyline-, convex-, and hybrid-layer approaches, and discuss how our proposed approach complements the state-of-the-art methods. (1) **Skyline layer approach:** DG [4] finds skylines iteratively and materializes them as layers. The dominance relationships between layers enable selective access to layers. However, for high-dimensional or anti-correlated data with a large set of skylines, the cardinality of each layer is high and the performance starts to deteriorate. Our dual-resolution layer indexing addresses this limitation by further splitting each layer into fine-granularity sublayers. (2) **Convex layer approach:** Onion [2] materializes convex skylines into layers. Since dominance relationships do not exist between adjacent layers, Onion requires complete access to layers, and thus incurs higher access costs than DG. AppRI [3] improves Onion by designing a *robust* index, which reduces the sizes of tuples in each layer by counting the number of dominating tuples. However, AppRI cannot avoid complete access to layers either. In clear contrast, our dual-resolution layer defines the dominance relationships between the coarse- and fine-granularity layers, and makes use of those relationships to enable selective access to all layers. (3) **Hybrid layer approach:** HL [5] materializes convex skyline as layers, but builds d sorted lists within each layer for d attributes. This sorted ordering helps avoid unnecessary access within a layer, and thus enables selective access. However, compared to our systematic approach of defining and leveraging formal relationships, the selectivity is one order of magnitude higher (as reported in Section 8).

9.2 List-Based Approach

The list-based approach exploits a set of sorted lists for efficient top-k processing. FA [7], TA [10], MPro [11], Upper [12], and Unified [14] are well-known algorithms in this approach. Given a scoring function $\mathcal{F}$, these algorithms access and aggregate the score of tuples by iteratively accessing the sorted lists in a round-robin manner until the best k tuples seen become the top-k answers. However, as sorted lists may have different importance (or weights) and may also be correlated, round-robin access may not be optimal. Some research work [15] has been carried out optimal access scheduling of sorted lists. However, the layer-based approaches are inherently robust to weights and correlations, and do not require such scheduling.

9.3 View-Based Approach

The view-based approach examines pre-computed top-k queries as *views* in database systems. The key idea underlying these approaches such as PREFER [16] and LPTA [17] is to select the most similar materialized top-k query, and to reuse access and computation to compute a new top-k query. Specifically, PREFER determines a *watermark* tuple to obtain the top-k answers from the existing query. LPTA optimizes PREFER by combining multiple top-k queries using a linear programming technique. A drawback is the overhead of storing and managing multiple top-$k$views.

10 Conclusion

This paper has studied the problem of designing a layer-based index for supporting scalable top-k query computation. In particular, we focused on the optimization goal of making selective access to each layer. To pursue this optimization goal, we designed a dual-resolution layer, in which each coarse-level layer was further divided into multiple fine-level sublayers at a finer granularity. We then addressed various optimization directions to make the dual-resolution layer scalable. In our experiments, we validated that our proposed dual-resolution layer is scalable over extensive data settings.

Acknowledgments

This work was supported in part by the ICT Research and Development program of MOST/KOSEF under Grant 10041244. This paper builds on and significantly extends our preliminary work [1]. S. Hwang is the corresponding author.

References

[1] J. Lee, H. Cho, and S.-W. Hwang, "Efficient dual-resolution layer indexing for top-k queries," in *Proc. IEEE 28th ICDE*, Washington, DC, USA, 2012, pp. 1084–1095.

[2] Y.-C. Chang *et al.*, "The onion technique: Indexing for linear optimization queries," in *Proc. ACM SIGMOD*, Dallas, TX, USA, pp. 391–402, 2000.

[3] D. Xin, C. Chen, and J. Han, "Towards robust indexing for ranked queries," in *Proc. 32nd Int. Conf. VLDB*, Seoul, Korea, 2006, pp. 235–246.

[4] L. Zou and L. Chen, "Dominant graph: An efficient indexing structure to answer top-k queries," in *Proc. IEEE 24th ICDE*, 2008, pp. 536–545.

[5] J. Heo, J. Cho, and K. Whang, "The hybrid-layer index: A synergic approach to answering top-k queries in arbitrary subspaces," in *Proc. IEEE 26th ICDE*, Long Beach, CA, USA, 2010, pp. 445–448.

[6] L. Zou and L. Chen, "Pareto-based dominant graph: An efficient indexing structure to answer top-k queries," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 5, pp. 727–741, May 2011.

[7] R. Fagin, "Combining fuzzy information from multiple systems," in *Proc. 15th PODS*, Montreal, QC, Canada, 1996, pp. 216–226.

[8] S. Nepal and M. V. Ramakrishna, "Query processing issues in image (multimedia) databases," in *Proc. 15th ICDE*, Sydney, NSW, Australia, 1999, pp. 22–29.

[9] U. Güntzer, W.-T. Balke, and W. Kießling, "Optimizing multi-feature queries for image databases," in *Proc. 26th Int. Conf. VLDB*, Cairo, Egypt, 2000, pp. 419–428.

[10] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 614–656, 2003.

[11] K. C. Chang and S. Hwang, "Minimal probing: Supporting expensive predicates for top-k queries," in *Proc. 2002 ACM SIGMOD*, Madison, WI, USA, pp. 346–357.

[12] N. Bruno, L. Gravano, and A. Marian, "Evaluating top-k queries over web-accessible databases," in *Proc. ICDE*, 2002, pp. 369–378.

[13] M. Theobald, G. Weikum, and R. Schenkel, "Top-k query evaluation with probabilistic guarantees," in *Proc. 30th Int. Conf. VLDB*, Toronto, ON, Canada, 2004, pp. 648–659.

[14] S. Hwang and K. C. Chang, "Optimizing top-k queries for middleware access: A unified cost-based approach," *ACM Trans. Database Syst.*, vol. 32, no. 1, pp. 1–41, 2007.

[15] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum, "IO-top-k: Index-access optimized top-k query processing," in *Proc. 32nd Int. Conf. VLDB*, Seoul, Korea, 2006, pp. 475–486.

[16] V. Hristidis, N. Koudas, and Y. Papakonstantinou, "PREFER: A system for the efficient execution of multi-parametric ranked queries," in *Proc. 2001 ACM SIGMOD*, pp. 259–270.

[17] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis, "Answering top-k queries using views," in *Proc. 32nd Int. Conf. VLDB*, Seoul, Korea, 2006, pp. 451–462.

[18] J.-S. Heo, K.-Y. Whang, M.-S. Kim, Y.-R. Kim, and I.-Y. Song, "The partitioned-layer index: Answering monotone top-k queries using the convex skyline and partitioning-merging technique," *Inf. Sci.*, vol. 179, no. 19, pp. 3286–3308, 2009.

[19] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proc. 17th ICDE*, Berlin, Germany, 2001, pp. 421–430.

[20] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: An online algorithm for skyline queries," in *Proc. 28th Int. Conf. VLDB*, Hong Kong, 2002, pp. 275–286.

[21] J. Matousek and O. Schwarzkopf, "Linear optimization queries," in *Proc. Symp. Comput. Geom.*, Berlin, Germany, 1992, pp. 16–25.

[22] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry*. Berlin, Germany: Springer, 2008.

[23] B. Chazelle, "On the convex layers of a planar set," *IEEE Trans. Inf. Theory*, vol. 31, no. 4, pp. 509–517, Jul. 1985.

[24] A. Soffer *et al.*, "Static index pruning for information retrieval systems," in *Proc. 24th SIGIR*, New Orleans, LA, USA, 2001, pp. 43–50.

[25] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 41–82, 2005.

[26] A. Vlachou, C. Doulkeridis, and Y. Kotidis, "Angle-based space partitioning for efficient parallel skyline computation," in *Proc. ACM SIGMOD*, Vancouver, BC, Canada, 2008, pp. 227–238.

[27] J. Lee and S. Hwang, "BSkyTree: Scalable skyline computation using a balanced pivot selection," in *Proc. 13th Int. Conf. EDBT*, Lausanne, Switzerland, 2010, pp. 195–206.

[28] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Trans. Math. Softw.*, vol. 22, no. 4, pp. 469–483, 1996.

[29] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM CSUR*, vol. 40, no. 4, Article 11, 2008.

Jongwuk Lee is a Post-Doctoral Researcher with the College of Information Sciences and Technology, Pennsylvania State University, University Park, PA, USA. He received the Ph.D. degree in computer science and engineering at Pohang University of Science and Technology, Pohang, Korea, in 2012. His current research interests include query processing and optimization, information retrieval, data mining, and web mining.

Hyunsouk Cho is a Ph.D. student with Pohang University of Science and Technology (POSTECH), Pohang, Korea. He received the B.S. degree in computer science and engineering at POSTECH in 2011. His current research interests include advanced query processing and optimization in database systems.

Sunyou Lee is a Ph.D. student with Pohang University of Science and Technology (POSTECH), Pohang, Korea. She received the B.S. degree in electrical engineering at POSTECH in 2013. Her current research interests include top-*k* query processing in database systems.

Seung-won Hwang is an Associate Professor with the Department of Computer Science and Engineering at Pohang University of Science and Technology (POSTECH), Pohang, Korea. Prior to joining POSTECH, she received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA. Her current research interests include querying and mining entities extracted from structured and unstructured sources.