Pareto-Based Dominant Graph: An Efficient Indexing Structure to Answer Top-K Queries

Lei Zou, *Member*, *IEEE*, and Lei Chen, *Member*, *IEEE*

Abstract—Given a record set D and a query score function F, a top-k query returns k records from D, whose values of function F on their attributes are the highest. In this paper, we investigate the intrinsic connection between top-k queries and dominant relationships between records, and based on which, we propose an efficient layer-based indexing structure, Pareto-Based **D**ominant **G**raph (DG), to improve the query efficiency. Specifically, DG is built offline to express the dominant relationship between records and top-k query is implemented as a graph traversal problem, i.e., *Traveler* algorithm. We prove theoretically that the size of search space (that is the number of retrieved records from the record set to answer top-k query) in our algorithm is directly related to the cardinality of skyline points in the record set (see Theorem 3). Considering I/O cost, we propose cluster-based storage schema to reduce I/O cost in Traveler algorithm. We also propose the cost estimation methods in this paper. Based on cost analysis, we propose an optimization technique, pseudorecord, to further improve the search efficiency. In order to handle the top-k query in the high-dimension record set, we also propose N-Way Traveler algorithm. In order to handle DG maintenance efficiently, we propose "Insertion" and "Deletion" algorithms for DG. Finally, extensive experiments demonstrate that our proposed methods have significant improvement over its counterparts, including both classical and state art of top-k algorithms.

Index Terms—Top-k query, database, algorithms.

1 Introduction

Given a record set D and a query function F, a top-k preference query (*top-k query* for short) returns k records from D, whose values of function F on their attributes are the highest. Due to the popularity of top-k queries, there are many solutions that have been proposed in the database literature. They can be divided into three categories: *Sorted list-based*, *Layer-based* and *View-based*.

1. *Sorted lists-based*. The methods in this category sort the records in each dimension and aggregate the rank by parallel scanning each list until the top-k results are returned, such as TA and CA [1], [2], [3].
2. *Layer-based*. The algorithms in this category organize all records into consecutive layers, such as Onion [4] and AppRI [5]. The organization strategy is based on the common property among the records, such as the same convex hull layer in Onion [4]. Any top-k query can be answered by up to k layers of records.
3. *View-based*. Materialized views are used to answer a top-k query, such as PREFER [6] and LPTA [7].

The algorithms in the first category need to merge different sorted lists, and their query performance is not as good as the rest two categories. However, the layer-based and view-based approaches have some common limitations: 1) most existing algorithms in these two categories only support linear query functions, such as [4], [5], [6], [7]; and 2) they have high maintenance cost. Observed from the limitations, we propose a novel layer-based method in this paper.

Parallel to top-k queries, there is another type of query, *skyline query*, which also attracts much attention recently [8], [9], [10], [11]. Given a record set D of dimension m, a skyline query over D returns a set of records which are not dominated by any other record. Here, we say one record X dominates another record Y if among all the m attributes of X, there exists at least one attribute value is smaller than the corresponding attribute in Y and the rest the attributes of X are either smaller or equal to the matched attributes in Y. The difference between skylines and top-k queries is that the results of top-k queries change with the different preference functions, while the results of skyline queries are fixed for a given record set. Many solutions have been proposed for computing skylines, such as block nested loops [8], divide-and-conquer [8], bitmap [10], index [10], nearest neighbor [11], and branch and bound [9].

In fact, a top-k and a skyline queries are not independent to each other, they are connected by the dominant relationships. Therefore, we would like to make use of dominant relationship to answer a top-k query. The intuition is that for the aggregate monotonic function, F, of a top-k query, when record X dominates record Y, the query score of X must be less than that of Y. It also means that X can be pruned safely when we know Y is not in the top-k answer. Therefore, top-k answers can be derived from *offline* generated dominant relationships among all the records.

In this paper, with the consideration of the intrinsic connection between a top-k problem and dominant relationships, we design an efficient indexing structure, Pareto-Based **D**ominant **G**raph (DG), to support a top-k query. We derive the most important property in DG: *the necessary*

- *L. Zou is with the Institute of Computer Science and Technology, Peking University, No. 5 Yiheyuan Road Haidian District, Beijing 100871, China. E-mail: zoulei@icst.pku.edu.cn.*
- *L. Chen is with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China. E-mail: leichen@cse.ust.hk.*

Manuscript received 30 Aug. 2008; revised 14 May 2009; accepted 28 Sept. 2009; published online 18 Nov. 2010.
Recommended for acceptance by S. Chakravarthy.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2008-08-0450.
Digital Object Identifier no. 10.1109/TKDE.2010.240.

condition that a record can be in top-k answers is that all its parents in DG have been in top-(k-1) answers (see Property 1). Benefiting from the property, the search space (that is the number of retrieved records from the record set to answer a top-k query) of a top-k query is reduced greatly. We prove theoretically that *the size of search space in our basic algorithm is directly related to the cardinality of skyline points in the record set* (see Theorem 3). Based on cost analysis, we propose the optimization technique, *pseudorecord*, to further reduce the search space. Furthermore, a DG index can support any aggregate monotone query function (defined in Definition 1) and has the "light" maintenance cost. To summarize, in this paper, we make the following contributions:

1. We represent the dominant relationships among the records into a Pareto-Based *partial order* graph, called ***D**ominant **G**raph*. In a DG index, records are linked together according to dominant relationships. Based on offline built the DG, we propose *Traveler* algorithm to answer a top-k query, which transforms a top-k query to a graph traveler problem.
2. We propose an analytic model for the cost estimation of Traveler (including both CPU cost and I/O cost). In order to estimate the cost, we propose several novel methods to estimate both CPU and I/O cost.
3. To improve query performance, we introduce a novel concept, *pseudorecord*, and extend *Traveler* to its advanced version, called Advanced *Traveler* algorithm, by utilizing the pseudorecords in the extended dominant graph. In order to handle high-dimensional data sets, we propose the N-Way *Traveler* algorithm for a top-k query. Based on I/O cost analysis, we propose cluster-based storage schema to store the records in each layer of the DG Index. Furthermore, we prove I/O optimality in our proposed Traveler algorithm.
4. In order to handle deletion and insertion, we design efficient DG index maintenance algorithms. We also address maintenance issues in the cluster-based storage schema.
5. We show by extensive experiment results that our methods have significant improvement over both classical and state art of top-k algorithms. For example, the search space in our algorithm is less than $\frac{1}{5}$ of that in AppRI [5], one of state art of top-k algorithms. Due to the optimization technique, pseudorecord, Advanced *Traveler* algorithm still works well in the worst case, that is, no dominant relationship exists in the original record set.

The remainder of the paper is organized as follows: DG and basic *Traveler* algorithm (a memory-based version of Traveler algorithm) for top-k query are proposed in Section 2. Considering I/O cost, we propose Traveler algorithm in Section 3. We discuss cost estimation methods in Section 4. In order to further reduce search space, we introduce "pseudorecords" and extend Traveler algorithm to its advanced version in Section 5. We also propose the solution to handle high dimensions. We address maintenance issues in Section 6. We evaluate the efficiency of our DG approach with extensive experiments in Section 7. Finally, Section 9 concludes the paper.

TABLE 1
Meanings of Symbols Used

D	Record Set
$\|D\|$	Cardinality of Record Set
$\|m\|$	Number of Dimension
r	Record
DG	Dominate Graph
L_i	the ith layer of DG
F	Query Function
CL	Candidate List
RS	Result Set
I_i	the ith dimension in record
$L_i.size$	Number of Records in the ith layer
$\|record\|$	bytes of each record
$\|page\|$	bytes of each disk page
$Parent(r)$	The set of r's parents

2 Top-k Query Based on DG

Definition 1 (Aggregate Monotone Function [1]). *An aggregation function F is a monotone if $F(x_1 \dots x_n) \leq F(x'_1 \dots x'_n)$, whenever $x_i \leq x'_i$ for every i.*

In this paper, we only consider top-k queries with aggregate monotone query functions. Intuitionally, if we can prerank all records using dominant relationships in offline processing, we can answer an online top-k query by traversing the "preranked list." Table 1 lists commonly used symbols in this paper.

2.1 Basic Traveler Algorithm

In this section, we propose a memory-based algorithm, called basic *Traveler*, to answer top-k query. In basic Traveler, we assume that all index and data files are maintained in memory.

Definition 2 (Dominate [12]). *Given two records r and r' in a multidimension space, we say that r* dominates *r' if and only if they satisfy the following two conditions: 1) in any dimension I_i, the value of r is larger than or equal to r', i.e., $r.I_i \geq r'.I_i$; 2) there must exist at least one dimension I_j, $r.I_j > r'.I_j$.*

Following other references, such as [13] and [14], we also call "dominate" as "Pareto-Based dominate." Since query functions are aggregate monotone functions, if record r dominates another record r', the score of r must be larger than r'. Note that Definition 2 about *dominate* is different from its counterparts in skyline [8], since we use relationship $>$ (or $\geq$) instead of $<$ (or $\leq$). However, they are essentially equivalent to each other. We do not distinguish them in this paper when the context is clear.

Definition 3 (Maximal Layers [12]). *Given a set S of records in a multidimension space, a record r in S that is dominated by no other records in S is said to be maximal. The first* maximal *layer L_1 is the set of maximal points of S. For $i > 1$, the ith maximal layer L_i is the set of maximal records in $S - \bigcup_{j=1\dots i-1} L_j$.*

Definition 4 (Dominant Graph). *Given a set S of records in a multidimension space, S has k nonempty maximal layers L_i, $i = 1 \dots n$. The records r in ith maximal layer and records r' in (i+1)th layer form a bipartite graph g_i, $i = 1 \dots (m-1)$. There is a directed edge from r to r' in g_i if and only if record r dominates r'. We call the directed edge as "parent-children*

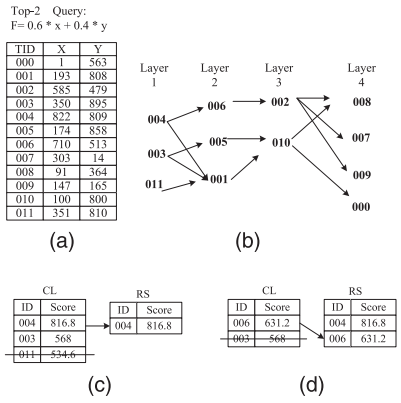


Fig. 1. Running example. (a) A Database D. (b) DG of the database D. (c) Step 1. (d) Step 2.

relationship" in the later discussion. All bipartite graphs g_i are joined to obtain Dominate Graph *(DG for short). The maximal layer L_i is called ith layer of DG.*

Building DG. To build a DG offline, we can use any skyline algorithm to find each layer of DG. Then, we build the "parent-children relationship" between records in the ith layer and that in the $(i+1)$th layer.

Essentially, DG is a lattice structure, which uses "dominating relationship" as "partial order," as defined in [15]. Note that DG is stored independently as the indexing structure for the record set. A DG index is shown in the following example.

Example 1. (Running Example) We have a record set D, as shown in Fig. 1a. The DG of the record set is shown in Fig. 1b. Given a query function $\mathrm{F} = 0.6 * \mathrm{X} + 0.4 * \mathrm{Y}$, we want to answer top-2 query.

Fig. 1 shows the DG index for the running example. In DG, record 004 is a parent of 006. In fact, a record r may have several children (e.g., 004), and also may have several parents (e.g., 001).

Lemma 1. *Given a top-k query function F over record set D, if $r \in D$ and r is a parent of r' ($r' \in D$), and r' is in top-k answers, r must be in top-$(k-1)$ answers.*

Proof. According to Definition 4, we know that r dominates r', if r is a parent of r' in DG. Since the query function F is a monotone aggregate function, the value of *F(r)* must be larger than $F(r')$. Therefore, if $F(r')$ is in top-k answers, $F(r)$ must be in top-$(\mathrm{k}-1)$ answers. □

Based on Lemma 1, we can convert a top-k query into a graph traversal problem in DG. The basic *Traveler* algorithm is presented in Algorithm 1.

In basic *Traveler*, first, we compute scores for all records in the first layer of DG according to the query function F and insert them into the max-heap Candidate List (i.e., *CL*) in nonascending order of scores (Line 1 in Algorithm 1). We keep the size of CL no larger than k, since we only need to report top-k answers. Then, we move the record with the largest score from CL to the answer set RS (Line 2). Next, in Lines 5-13, we find the $(n+1)$th largest answer in the nth iteration step. Assume that the nth largest record is denoted as r, which is obtained in the previous iteration step. In Lines 5-9, for each child C of record r, if C has another parent that is not in RS now, C will not be assessed in this step (Line 7) (the correctness will be proved later). Otherwise, we compute the score of C and insert it into CL (Line 9). In addition, if some child C has been computed using query function before, C will be ignored (Line 6). At the end of the nth iteration step, it is only necessary to maintain the first $k-n$ records (Lines 10-11). The record with the largest score in CL now must be the $(\mathrm{n}+1)$th largest answer, which is moved from CL to RS (Line 12). The above processing (Lines 5-13) is iterated until we find the correct top-k answers. Then, we report them (Line 14). In the running example, since record 003, 004, and 011 are in the first layer of DG, they are computed by query function. They are inserted into CL, as shown in Fig. 1c. Since we need to answer top-2 query, we keep the size of CL to be 2. Therefore, we delete 011 from CL. 004 is the largest, and it is moved from CL to RS. 006 is 004's child, and it is computed and inserted into CL. 001 is also 004's child. But 001 has another parent 011, which is not in RS now. Therefore, 001 is not considered. The largest record, that is 006, is moved from CL into RS, as shown in Fig. 1d. Now, top-2 answers have been found in RS. The algorithm terminates.

Algorithm 1 basic Traveler Algorithm (basic *Traveler*)

Require: Input: a DG of the database D and top-k query function F.
Output: the top-k answers
1: All records at 1st layer of DG are computed by query function F. They are inserted into the sorted queue Candidate List (i.e. CL), whose size $\leq$ k.
2: Move the first largest record r in CL into result set RS.
3: set n=1;
4: **while** $(RS.size() < k)$ **do**
5: **for** each child C of r **do**
6: **if** C has at least one parent that is not in RS or C has been computed using query function F before **then**
7: Continue
8: **else**
9: Compute C by query function F and insert it into CL.
10: **if** the size of CL is larger than k- n **then**
11: Only keep the first k-n records in CL
12: Move the first largest record r in CL into the answer set RS.
13: n = n+1
14: Report RS

2.2 Correctness of Basic Traveler

Theorem 1. *Basic Traveler Algorithm can find the correct top-k answers.*

Proof. (Sketch) (proof by induction)

1. (*Base Case.*) For a record r' in the ath layer ($\mathrm{a} > 1$), we must find a record r in the first layer dominating r' according to DG definition. It means that the score of r is larger than that of r'. It also indicates that the top-1 answer must be in the first layer of DG. According to Line 1 in basic *Traveler*, we can find the correct top-1 answer.
2. (*Hypothesis.*) Assume that we have obtained the correct top-n answers ($n < k$) now, and the nth largest answer is record r, which is moved from *CL* to *RS*.

   (*Induction.*) For each child C of r, if C has another parent P' that is not in *RS*, it means that P' is not in top-n now. According to Lemma 1, it also indicates that C cannot be in top-$(\mathrm{n}+1)$. Therefore, only when all C's parents are in *RS* in the nth iteration step, C has the chance to be in

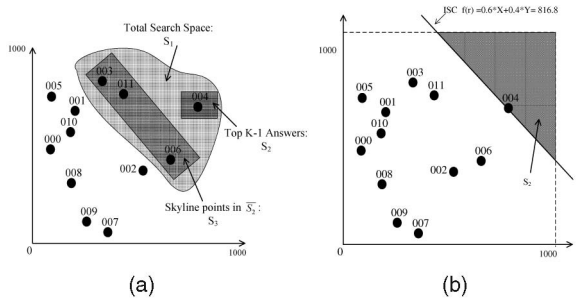


Fig. 2. Cost analysis. (a) Search Space in Traveler. (b) Identical Score Curve.

top-$(n+1)$ answers, that is, C will be computed and inserted into *CL* (Lines 7-12). Therefore, we can obtain the correct $(n+1)$th largest answer in the nth iteration step.

3. According to the above analysis, we can always obtain the correct top-$(n+1)$ answers, once we have obtained the correct top-n answers. Furthermore, we have proved that we can obtain the correct top-1 answer in *Base Case*. Therefore, we can find the correct top-k answers in basic *Traveler* algorithm. □

2.3 CPU Cost Analysis

Definition 5. *CPU cost of basic Traveler algorithm is the number of records to be computed by query function during top-k query.*

Property 1. *The necessary condition for a record r to be in top-k answers is that r has no parents in DG or all r's parents are in final top-(k-1) answers.*

Proof. There are two cases to be considered.

1. r has no parents: It means that r is in the first layer of DG. Thus, r will be accessed.
2. r has at least one parent: (proof by contradiction) Assume that there exists a record r', it has at least a parent P that is not in top-$(k-1)$ answers and r' is in top-k answers. Based on Lemma 1, P must be in top-$(k-1)$ answers, which is contradicted to the assumption. □

According to Line 6 in basic *Traveler*, the following lemma holds.

Lemma 2. *A record r in the ath $(a > 1)$ layer needs to be computed in Traveler algorithm, **if and only if** r's all parents are in final top-$(k-1)$ answers.*

In order to analyze the cost in basic Traveler, first, we define set $S_1 = \{r_i | r_i \text{ is computed by query function in basic } Traveler\}$. S_1 denotes the total search space in the algorithm. Obviously, according to Definition 5, the cost of basic Traveler equals to $|S_1|$, that is $Cost = |S_1|$. To evaluate $|S_1|$, we define two other sets: $S_2 = \{r_i | r_i \text{ is in the final top-}(k-1) \text{ answers}\}$. All records in the original database D except for those in S_2 are collected to form $\overline{S_2}$. $S_3 = \{r_i | r_i \in \overline{S_2}$ and there exist no records in $\overline{S_2}$ dominating r_i, namely, r_i is a skyline point in $\overline{S_2}\}$. In Fig. 2a, we show the total search space S_1 in the running example. There is only one record 004 in S_2, since only 004 is in top-$(k-1)$ answers ($k = 2$ in running example). Remove 004 from the record set. There are three skyline points in the left records, which form the set S_3. We prove theoretically that the following theorem holds:

Theorem 2. $S_1 = S_2 \cup S_3$.

Proof (Proof by Contradiction).

1. Assume that $\exists r',\ r' \notin (S_2 \cup S_3) \wedge r' \in S_1$

$$r' \notin (S_2 \cup S_3) \rightarrow (r' \in \overline{S_2}) \wedge (r' \in \overline{S_3}).$$

Observed from the above equation, r' is not in the final top-$(k-1)$ results, since r' is in $\overline{S_2}$. Because $r' \in \overline{S_3}$, there exists at least one record r in $\overline{S_2}$ dominating r'. It indicates that, for record r', it has at least one parent r that is not in final top-$(k-1)$ answers. According to Lemma 2, r' cannot be computed by query function in $Traveler$ algorithm, namely, $r' \notin S_1$, which is contradicted to the assumption.

2. Assume that $\exists r', r' \in (S_2 \cup S_3) \wedge r' \notin S_1$.

If $r' \in S_2$, r' must be in answer set RS. According to Algorithm 1, there exist no records r that is inserted into RS without being computed by query function. It indicates that r' must be computed, i.e., $r' \in S_1$. If $r' \in S_3$, r' is in the first layer or in ath $(a > 1)$ layer. If r' is in first layer, r' must be computed according to Line 1 in Algorithm 1, namely, $r' \in S_1$. If r' is in ath $(a > 1)$ layer, since r' is skyline point in $\overline{S_2}$, it is clear to know that all parents of r' are in top-$(k-1)$ answers. According to Lemma 2, r' must be computed, namely, $r' \in S_1$.

Therefore, $r' \in S_1$, which is contradicted to the assumption. Thus, according to 1) and 2), $S_1 = S_2 \cup S_3$. □

Theorem 3. *Given a record set D and a top-k query, the CPU cost of basic Traveler algorithm can be evaluated as follows:*

$$CPU\_Cost = k - 1 + |skyline(\overline{S_2})|,$$

where $|skyline(D)|$ is the cardinality of skylines in D.

Proof. According to Theorem 2, it is clear to know $CPU\_Cost = |S_1| = |S_2| + |S_3| = k - 1 + |skyline(\overline{S_2})|$. □

3 Traveler Algorithm

In Section 2.1, we propose memory-based Traveler algorithm, that is basic Traveler algorithm. In practice, we cannot assume that all data files and index files can be maintained in memory. Therefore, I/O cost is another key issue in query performance.

Definition 6. *I/O cost is defined as the number of I/O times during query processing.*

In order to reduce I/O cost, we propose our storage schema and disk-based Traveler algorithm (*Traveler* algorithm for short). As we know, disk page is the basic I/O unit. In the following analysis, without loss of generality, we assume that there is only one page swapping in each I/O.

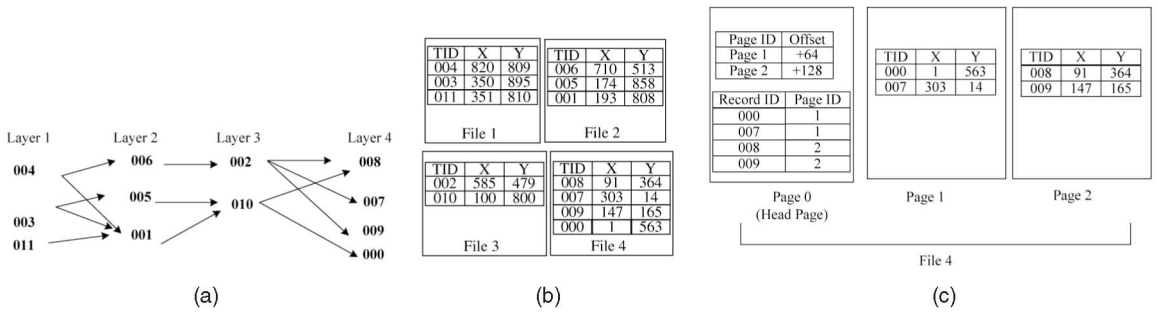


Fig. 3. Storage framework. (a) Index file. (b) Data file. (c) Page arrangement in each file.

Assume that we need to access $|N|$ records to answer top-k query. If these $|N|$ records are in the same disk page, I/O cost is only 1 (the optimal case). However, if these $|N|$ records are distributed in $|N|$ different disk pages, I/O cost is $|N|$ (the worst case). Obviously, different storage schemas will lead to different performances. Therefore, we first discuss our storage framework.

The original records and DG are stored separately. For example, the storage framework of the running example is shown in Fig. 3. All record IDs and the topology information and layer number of the dominant graph DG are stored as an index file. All records in each layer, including IDs and attribute values, are stored in a flat data file, which consists of several consecutive disk pages. Each flat file corresponds to one layer in DG. We record the pointers to the starting and ending pages of each small data file. All data files are collected to form the original database. In addition, there is a head page in each data file, which records following information: 1) the offset of each disk page in this data file; 2) pairs <record ID, pageID> are collected to form a hash table. For example, the head page of File 4 in Fig. 3 is shown in Fig. 3c. Assume that each page can contain at most two records and the size of each page is 64 bytes. Benefiting from the head page, given a record ID in this file, we can retrieve the specified record with twice disk I/O at most. In fact, the index file can be considered as an auxiliary indexing structure for the flat data files. In practice, we can cache some first layers of DG indexes in main-memory. When we perform top-k query ($k \leq 100$), the experiment results in Section 7.3 show that we only need to access some first layers. Therefore, in this work, we mainly study the I/O cost affected by data files, and ignore the I/O cost affected by DG indexes.

As shown in Fig. 3c, all records in the ath layer are stored in $\frac{L_i.size*|record|}{|page|}$ pages, where $L_i.size$ is the number of records in the ith layer and $|record|$ is the number of bytes in each record and $|page|$ is the number of bytes in each disk page. Given a top-k query function F, the records that have high possibilities to be accessed together should be put into one or a few disk pages.

To propose an effective storage strategy for records to reduce I/O cost, we need to address the following two key issues in turn: 1) How to measure the probability of two records r and r' in the same layer will be accessed together during the *Traveler* algorithms; 2) an effective cluster method: Based on the above possibility, we propose an efficient technique to arrange records of the same layer into different disk pages with aim to reduce I/O cost.

Key Issue 1. Given two records r and r' in the ath layer (a > 1), and a top-k query Q, how to measure the possibility that record r and r' will be accessed together in top-k query.

Algorithm 2 Disk-Based Traveler (Traveler for short)

Require: Input: a DG of the database D and top-k query function F.
Output: the top-k answers
1: Load all disk pages in the first layer of DG into memory.
2: All records in these disk pages are computed by query function F. They are inserted into the sorted queue Candidate List (i.e. CL), whose size $\leq$ k.
3: Move the first largest record r in CL into result set RS.
4: set n=1;
5: Initialize the set S to be empty.
6: **while** $(RS.size() < k)$ **do**
7: Initialize the set S' to be empty.
8: **for** each child C of r **do**
9: **if** C has at least one parent that is not in RS or C has been computed using query function F before **then**
10: Continue
11: **else**
12: Insert the child C into the set S'.
13: $S = S \cup S'$
14: Assume that the records in S' are stored in N disk pages P_i $(i = 1, ..., N)$.
15: Insert these N disk page IDs into heap H in decreasing order of the maximal score $MaxScore(P)$.
16: **while** true **do**
17: set β to be the first largest answer in candidate set CL
18: the head of the heap H is disk page P.
19: **if** $MaxScore(P) \leq \beta$ **then**
20: Break
21: Load the disk page P into memory
22: **for** each record c in P **do**
23: **if** $c \in S$ **then**
24: Compute query function value of c, and insert it into answer set CL.
25: **if** the size of CL is larger than k- n **then**
26: Only keep the first k-n records in CL
27: Move the first largest record r into answer set RS
28: $n = n + 1$
29: Report RS

In order to address the issue, we assume that two data records r and r' will be accessed together if their parent sets are similar. We explain the validity of the new assumption as follows: Given two records r and r', their parent sets are denoted as $Parent(r)$ and $Parent(r')$. In Traveler algorithm, the necessary condition that one record r (or r') needs to be assessed is that all records in $Parent(r)$ (or $Parent(r')$) have been accessed by now. If all records in $Parent(r)$ (or $Parent(r')$) have been visited and $Parent(r)$ is similar with $Parent(r')$, record r' (or r) has the high possibility to be accessed. Therefore, if $Parent(r)$ and $Parent(r')$ are similar with each other, r and r' have the high possibility to be accessed together. Obviously, the larger $Parent(r) \cap Parent(r')$ is, the more similar the two sets are. Therefore, we define "record distance" as follows:

Definition 7 (Record Distance). *Given two records r and r', the distance between r and r' is defined as follows:*

$$Dis(r,r') = |Parent(r) \cup Parent(r')| - |Parent(r) \cap Parent(r')| = |Parent(r) - Parent(r')| + |Parent(r') - Parent(r)|.$$

Obviously, the smaller $Dis(r,r')$ is, the more similar $Parent(r)$ and $Parent(r')$ are. It means that r and r' have the high possibility to be accessed together.

Key Issue 2. Given $L_a.size$ records in the ath layer, how to arrange these records in order to reduce I/O cost? Basically, we can cluster them into $\frac{L_i.size*|record|}{|page|}$ groups, and all records in one group are arranged into the same page. Obviously, each group has at most $\frac{|page|}{|record|}$ records.

Actually, we can use bottom-up "hierarchy" cluster method to solve Key Issue 2. At the beginning, all records are clustered into small groups using *Record Distance*, defined in Definition 7. Iteratively, based on Group Distance (defined in 8), some small groups are clustered into a larger group until that the number of records in a group is approximate to (no more than) $\frac{|page|}{|record|}$.

Definition 8 (Group Distance). *The distance between two groups G and G' is defined as the Hamming Distance between two sets $Parent(G)$ and $Parent(G')$, i.e., $|Parent(G) - Parent(G')| + |Parent(G') - Parent(G)|$, where $Parent(G) = \bigcup\{Parent(r_i)|r_i \in G\}$.*

All records in one group are stored in one disk page. For each disk page P, we define the upper corner and lower corner as follows, which will be used in Traveler algorithm (Algorithm 2):

Definition 9 (Upper Corner and Lower Corner). *Given a disk page P, the upper corner and lower corner of P are defined as follows:*

$$P^+ =< Max(I_1), \ldots, Max(I_n) >,$$
$$P^- =< Min(I_1), \ldots, Min(I_n) >,$$

where $Max(I_i) = Max\{r.I_i | r$ is a record in disk page $P\}$ and $Min(I_i) = Min\{r.I_i | r$ is a record in disk page $P\}$.

Notice that P^+ and P^- are stored in an index file. Since query function F is an aggregate monotone function, the maximal (or minimal) score in one disk page P can be evaluated as follows:

$$MaxScore(P) = F(P^+); MinScore(P) = F(P^-);$$

To reduce I/O cost, we propose disk-based Traveler algorithm as follows: At the beginning, we need to access all disk pages in the first layer of DG index (Line 1). All records in these disk pages are computed by query function F (Line 2). The largest record is moved into answer set RS (Line 3); and the others are inserted into candidate set CL. Before iteration, we initialize the set S to be empty. In the nth iteration of Lines 6-28, we find the $(n+1)$th largest answer in top-k query. We assume that record r is the nth largest answer, which is obtained in the previous iteration. Then, we consider each child C of record r. Similar with basic Traveler algorithm, if C has at least one parent that is not in RS or C has been computed using query function, we will ignore C in this step (Lines 9-10). Otherwise, we insert ID of record C into set S'. Assume that the records in S' are stored in N disk pages P_i $(i = 1, \ldots, N)$. We insert these disk page IDs into the heap H in decreasing order of $MaxScore(P_i)$ (Line 15). In the iteration of Lines 16-26, we always pop the heap head P. If $\beta > MaxScore(P)$, where β is the first largest answer in candidate set CL by now, it means that all unaccessed disk pages (in heap H) cannot contain the $(n+1)$th largest answer. Thus, the iteration (Lines 16-26) is terminated. Otherwise, we load disk page P into memory, where P is the head in heap H (Line 21). For each record c in P, if $c \in S$, it means that c needs to be computed by query function F. Actually, set S contains all records that are computed by query function F in basic Traveler algorithm. We compute $F(c)$ and insert it into candidate set CL. After the iteration between Lines 16-26, we move the largest record r in CL into answer set RS. We can guarantee that r must be the $(n+1)$th largest answer in top-k query. We iterate the above steps (Lines 6-28) until that there are k records in RS.

Theorem 4. *Given a query function F, the necessary condition that a disk page P needs to be loaded into memory is that $MaxScore(P) \geq \alpha$, where α is the final kth largest score.*

Proof. Assume that the kth largest record r is in page P. Given a disk page P', where $MaxScore(P') < \alpha$. Since r is in page P, it is straightforward to know $MaxScore(P) \geq \alpha$. Therefore, $MaxScore(P) > MaxScore(P')$. According to disk-based Traveler algorithm, we know P is at the head of P'. When P' is the heap head, P have been accessed. It means that record r has been in answer set RS, or r is the first record in candidate set CL. Since $MaxScore(P') < \alpha$, disk-based Traveler algorithm is terminated. Therefore, page P' will not lead to I/O cost. Therefore, Theorem 4 holds. □

4 Cost Estimation

In practice, the system should be able to estimate the running time of query algorithm in advance. For example, given multiple top-k query tasks over multicore system, we need to predict each task's running time to find the optimal task schedule. Therefore, we discuss how to estimate the cost in Traveler algorithm.

Definition 10. *Given a top-k query Q_k, the cost of Traveler algorithm is defined as follows:*

$$Traveler\_TotalCost(k) = CPU\_Cost * T_{cpu} + IO\_Cost * T_{IO}, \quad (1)$$

where T_{cpu} is the average cost (running time) for one computation, and T_{IO} is the average cost (running time) for one I/O, and CPU_Cost (defined in Definition 5) and IO_Cost (defined in Definition 6) are CPU cost and I/O cost, respectively.

4.1 Estimating CPU_Cost

It is straightforward to know CPU cost in Traveler algorithm is the same with that in basic Traveler algorithm. According to Theorem 3, $CPU\_Cost = k - 1 + |Skyline(\overline{S_2})|$. Therefore, in order to estimate CPU cost, we need to know the set $\overline{S_2}$ ($\overline{S_2}$ is defined in Section 2.3).

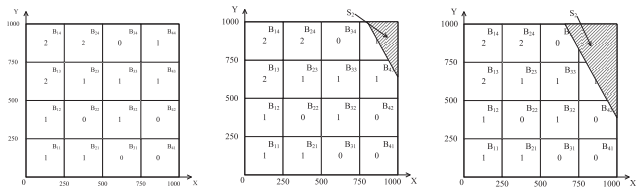


Fig. 4. Query cost estimation. (a) Equiwidth Histogram. (b) First Tested S_2. (c) Second Tested S_2.

However, it is not possible to know the exact value of $\overline{S_2}$ before we conduct Traveler algorithm. As a consequence, we cannot know the the cardinality of skylines in $\overline{S_2}$, (i.e., $|Skyline(\overline{S_2})|$). To overcome these issues, we use multidimensional histograms to estimate CPU cost. Specifically, we use the equiwidth histograms, which partition the d-dimensional data space into c^d regular buckets (where c is a constant called the histogram resolution). Similar with other histogram methods [16], we also assume that the distribution in each bucket is uniform. In this work, the number of buckets is determined by the available main-memory. As shown in Fig. 4a, we partition the whole data space of the running example into 4^2 buckets.

Assume that we know the (k-1)th largest query score is v. We can get a "identical score curve" (ISC) [16], which means the data points on the curve have exactly the same score (i.e., $f(r) = v$). An example of ISC is shown in Fig. 2b. The ISC divides the whole data space into two parts, "large" and "small," which, respectively, contain the records whose scores are strictly larger and smaller than v. Since v is the $(k-1)$th largest query score, it is straightforward to know that all records in the "larger" area are in top-$(k-1)$ answers. Therefore, the "larger" area is the same as the area S_2, as shown in Fig. 2b. With ISC, two key issues to estimate CPU cost in Traveler algorithm are: 1) How to estimate the ISC; 2) According to the estimated ISC, how to estimate the cardinality of skylines in $\overline{S_2}$. We address these two issues as follows:

We adopt the similar methods in [16] to find the estimated ISC. We sort the centroids of all buckets in decreasing order of their scores. In the running example, the sorted order is B_{44}, $B_{43}, \ldots$. First, we assume that the ISC passes the centroid of B_{44}, i.e., $f(r) = 875$. The shaded area in Fig. 4b shows the first tested S_2 area. Then, we estimate the number of records in S_2 (that is $|S_2|$). We inspect all buckets that intersect with S_2 (including the buckets that are contained completely in S_2). Specifically, for each bucket B, we compute the number of records in B that fall into S_2, denoted as $|B \cap S_2|$. Since the distribution of each bucket B is approximately uniform, $|B \cap S_2| = |B| \times area(B \cap S_2)/area(B)$, where $area(B \cap S_2)$ denotes the area of the intersection between B and S_2, and $area(B)$ denotes the area of B. Similar with [16], we also adopt Monte-Carlo methods to estimate $area(B \cap S_2)/area(B)$. Specifically, we first randomly generate a set of records inside the bucket (the number of generated records is α). Then, the number β of records falling in S_2 is counted. After that, $area(B \cap S_2)/area(B) \approx \frac{\beta}{\alpha}$. In Fig. 4b, the estimated number of records in S_2 is 0.52, which is smaller than $k - 1 = 1$. Therefore, we need to expand the tested S_2. According to the sorted bucket list, we examine the second bucket B_{43}, i.e., ISC crosses the centroid of B_{43}, as shown in Fig. 4c. Similarly, we estimate the number of records in the second tested S_2, that is 1.48, which is larger than $k - 1 = 1$. Therefore, we can stop expanding S_2.

In the next step, according to the estimated ISC, we need to estimate the cardinality of skylines in $\overline{S_2}$. In this paper, we utilize "sampling" technique to estimate $|Skyline(\overline{S_2})|$. Sampling techniques are often used in selectivity estimation [17] and query size estimation [18]. According to the analysis in [19], the cardinality of skyline points in the database D grows some power of $log|D|$, namely, $|Skyline(D)| \approx Alog^B|D|$, where A and B are two parameters. We compute the skyline on a small sample of the database D to estimate A and B, and then use them to calculate the size of skyline records in the whole database D.

In order to estimate $Skyline(\overline{S_2})$, we first randomly choose N records from the original database. We compute the query scores of these N records. Assume that there are only M records which are in the estimated $\overline{S_2}$. These M records are chosen as sampling records. We compute the skyline in these M records to estimate the parameters A and B. The number of sampling records are always very small. Therefore, we can adopt the main-memory-based algorithms to find skylines in sampling records. When we know the values of A and B, it is straightforward to compute $|Skyline(\overline{S_2})|$ by $Alog^B|\overline{S_2}|$. The total estimated CPU cost is $(k-1) + Alog^B|\overline{S_2}|$.

4.2 Estimating IO_Cost

As we know, IO_Cost is defined to be the number of I/O times, namely, the number of disk pages that we need to access in Disk-based Traveler algorithm. In order to estimate IO_Cost, we first estimate the final kth largest score, that is α. The estimated value of α is denoted as $\widehat{\alpha}$. We adopt the similar methods in Section 4.1 to obtain $\widehat{\alpha}$. Specifically, we use the equiwidth histogram and ISC method in Section 4.1. The only difference is that the "larger" area contains k records instead of $k - 1$ records in Section 4.1. According to estimated ISC, it is straightforward to obtain $\widehat{\alpha}$. According to Theorem 4, the estimated value of IO_Cost equals to be the number of disk pages P, where $Maxscore(P) \geq \widehat{\alpha}$.

5 Advanced Traveler Algorithm

5.1 Pseudorecords

According to Line 1 in Traveler algorithm, we need to compute all records in the first layer of DG. In order to reduce cost, we introduce *pseudorecords* to prune as many false candidates as possible. Notice that pseudorecords do not exist in original database. We only introduce them for pruning search space.

Given another database D and a top-2 query in Fig. 5a, different from the running example in Fig. 1, records 001-006 are all in the first layer of DG. We introduce some pseudorecords to dominate records in the first layer, and build an *Extended Dominate Graph* (Extended DG), as shown in Fig. 5b. According to basic *Traveler* algorithm, we can obtain the top-2 answers by only assessing record P003, P002, P001, 001, and 002, where record P003, P001, and P002 are introduced pseudorecords. Thus, after introducing pseudorecords, the cost is 5, smaller than 6 in basic *Traveler*. The above

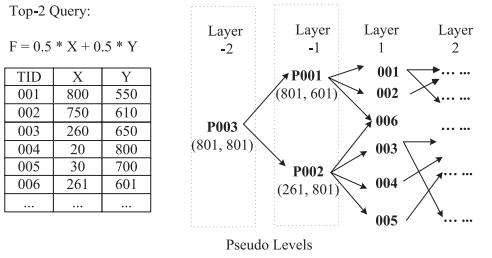


Fig. 5. Motivation example. (a) The database D. (b) Extended DG of the Database D.

motivation example in Fig. 5 indicates that pseudorecords are useful to reduce the search space. However, pseudorecords may lead to extra I/O cost. Therefore, we require that all extended layers can be maintained in main-memory. We introduce pseudorecords by the following method.

First, we build main-memory R-tree for the records in the first layer. Then, for each MBR M in R-tree, we introduce its upper corner (defined in Definition 11) as a pseudorecord. We introduce "parent-child" relationships among these pseudorecords and the records in the first layer to build extend layers. Since all extended layers are cached in main-memory, they will not lead to extra I/O cost.

Definition 11. *Given an MBR M in n-dimensional space, M's upper corner is defined as follows:*

$$M^+ =< Max(r.I_1), \ldots, Max(r.I_n) >,$$

where r is a record that is contained in MRB M.

5.2 Advanced Traveler

We extend Traveler algorithm to its advanced version in Algorithm 3. Algorithm 3 only shows the difference between Advanced Traveler algorithm and Traveler algorithm, which are Lines 6 and 26 (in Algorithm 2). In fact, the only difference between Advanced Traveler and Traveler is that: Advanced Traveler only counts real records (not pseudorecords) as shown in Lines 6 and 26. When we determine the size of CL, we also only count real records (Line 26).

Algorithm 3 Advanced Traveler

6) **While**(the number of real record* in RS < k)
... ...
26) **IF** the number of "real records" in CL is larger than k-n
...
*real records: not pseudo records.

Theorem 5. *Given a top-k query Q_k over the database D, if there are m pseudorecords whose scores are larger than the final kth answer, the computation cost of Advanced Traveler algorithm can be evaluated by the following equation:*

$$A\_Traveler\_TotalCost(k) = Traveler\_TotalCost(k+m), \quad (2)$$

where $Traveler\_TotalCost(k+m)$ can be evaluated by (1).

In order to estimate the cost in Advanced Traveler, we first need to estimate the value of m. After that, we can adopt methods in Section 4 to estimate the cost. We can first use histogram methods in Section 4 to estimate the final kth largest score, that is $\hat{\alpha}$. According to $\hat{\alpha}$ and query function f, we obtain ISC $f(r) = \hat{\alpha}$. Then, we still use histogram methods to estimate the number of pseudorecords (i.e., the value of m) in "larger" area. We omit details due to space limitation.

5.3 N-Way Traveler

In Algorithms 1 and 2, we need to access all records in the first layer of DG. As we know, as the number of dimensions increases, the number of skyline points becomes very large. It means that there are a large number of records in the first layer of DG in high-dimensional space. In order to address this issue, we propose N-Way Traveler algorithm.

Given an m dimension record set, if m is large, we divide the m dimensions into n sets. For each dimension set, we build DG, denoted by DG_i. Given a query function F, we decompose it into n subfunctions,[1] that are $F(x_1 \ldots x_m) = G(f_1(x_1 \ldots x_{d_1}) f_2(x_{d_1+1} \ldots x_{d_2}) \ldots f_n(x_{d_{n-1}+1} \ldots x_m))$, where I_i is a dimension. N-Way *Traveler* algorithm travels nDG_i in parallel until that all unassessed records cannot be in top-k answers. Traveling on each DG_i ranks records on the value of subfunction f_i. Only for presentation purpose, we discuss how to extend basic Traveler algorithm to N-Way Traveler algorithm. Actually, it is also straightforward to extend Traveler algorithm to N-Way version.

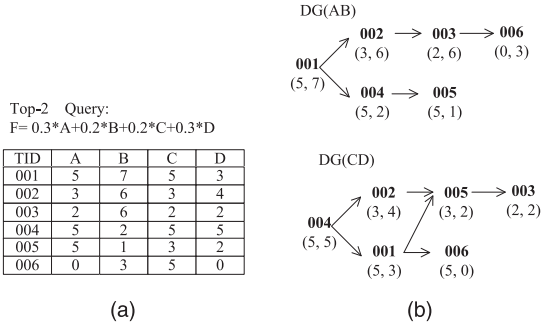
Algorithm 4 N-Way Traveler Algorithm

Require: Input: n-way DG on each dimension set I_i, i=1...n, and top-k query function F.
1: **for** each DG_i **do**
2: all records at 1st layer are computed by sub-function f_i on dimension set I_i. They are inserted the sorted heap CL_i.
3: The record r_i is computed by query function F and inserted into CL, where r_i is the record at the head of heap CL_i.
4: Update δ to be k-th largest score in CL.
5: Set $\beta = G(f_1(r_1|_{I_1}) \ldots f_n(r_n|_{I_n}))$, where $r_i|_{I_i}$ is the projection of r_i on dimension set I_i, and r_i is the record at the head of heap CL_i.
6: **while** ($\delta < \beta$) **do**
7: **for** each DG_i **do**
8: we move the heap head r_i from CL_i to another max-heap RS_i.
9: All r_i's children whose all parents are in RS_i now are collected to form the set C_i.
10: All records in C_i are computed by sub-function f_i on dimension set I_i. They are inserted into the heap CL_i.
11: The record r_i is computed by query function F and inserted into RS, where r_i is the record at the head of heap CL_i.
12: Set $\beta = G(f_1(r_1|_{I_1}) \ldots f_n(r_n|_{I_n}))$, where r_i is the record at the head of heap CL_i.
13: Update δ to be k-th largest score in CL.
14: Report top-k answers in CL.

In fact, we combine *TA* algorithm [1] and basic *Traveler* algorithm into the N-Way *Traveler*. The pseudocodes of N-Way *Traveler* are given in Algorithm 4. Initially, for each DG_i, the records in the first layer are computed by subfunction f_i and inserted into the max-heap CL_i (Line 2). These records are also computed by query function F and are inserted into the max-heap candidate list (i.e., CL) (Line 3). We always keep δ to be the kth largest score in CL by now (Lines 4 and 13). For each heap CL_i, we choose the heap head r_i. We keep $\beta = G(f_1(r_1|_{I_1}) \ldots f_n(r_n|_{I_n}))$, where $r_i|_{I_i}$ is the projection of r_i on dimension set I_i (Lines 5 and 12). It can be proved that β is the upper bound of the scores for all unassessed records. The algorithm is iterated until $\delta \geq \beta$ (Lines 7-13). During the iteration, for each DG_i, the operation is analogous to Traveler (Lines 8-11).

Example 2. The four-dimensional database is given in Fig. 6a. Given a query function $\mathrm{F} = 0.3 * \mathrm{A} + 0.2 * \mathrm{B} + 0.2 * \mathrm{C} + 0.3 * \mathrm{D}$, we want to report Top-2 answers.

1. We assume that query function F is decomposable.



TID	A	B	C	D
001	5	7	5	3
002	3	6	3	4
003	2	6	2	2
004	5	2	5	5
005	5	1	3	2
006	0	3	5	0

Fig. 6. Running example of N-Way Traveler algorithm. (a) A Database D. (b) N-Way DG Index.

First, we build two DG indexes for the four dimensions. The first one is built for "AB" dimensions and the second one is built for "CD" dimensions, which are shown in Fig. 6b. Furthermore, we divide the query function F into two subfunctions f_1 and f_2, where $f_1 = 0.3 * \mathrm{A} + 0.2 * \mathrm{B}$ and $f_2 = 0.2 * \mathrm{C} + 0.3 * \mathrm{D}$. Fig. 7 shows the steps in Example 2. In the first step, for "AB" dimensions, we compute $f_1(001) = 2.9$ and insert it into CL_1 (Line 2 in Algorithm 4). Since 001 is the largest in CL_1, we compute $F(001) = 4.8$ and insert it into CL (Line 3). Similarly, for "CD" dimensions, we compute $f_2(004) = 2.5$ and insert it into CL_2 (Line 2). We also compute $F(004) = 4.4$ and insert it into CL (Line 3), since 004 is the largest in CL_2. We set δ to be the second answer in CL, that is $\delta = 4.4$ (Line 4). We set $\beta = f_1(001) + f_2(004) = 5.4$, since 001 and 004 are the largest in CL_1 and CL_2, respectively (Line 5). $\delta < \beta$, the algorithm continues (Line 6). For "AB" dimensions, we move the record 001 from CL_1 into RS_1 (Line 8). Then, for "AB" dimensions, since 002 is a child of 001 and it has no other parents that are not in RS_1, we compute $f_1(002)$ and insert them into CL_1 (Lines 9-10). Due to the same reason, we compute $f_1(004)$ and insert them into CL_1(Lines 9-10). Since 002 is the largest one in CL_1 now, we compute $F(002) = 3.9$ (Line 11). Similarly, in "CD" dimensions, we compute $f_2(002)$ and $f_2(001)$ and insert them into CL_2 (Lines 9-10). We set $\beta = f_1(002) + f_2(001) = 4.0$ (Line 12). We update δ to be the second answer in CL, which is still 4.2 (Line 13). Since $\delta > \beta$, the algorithm terminates and report top-2 answers in CL, which are 001 and 004 (Line 14).

Theorem 6. *Given a top-k query Q_k over nDG_i by N-Way Traveler algorithm, if there are k_i records in max-heap RS_i in each DG_i, the cost of N-Way Traveler algorithm can be evaluated by the following equation:*

$$N\_Traveler\_TotalCost(k) = \sum_{i=1...n} Traveler\_TotalCost(k_i). \tag{3}$$

We can first use histogram methods in Section 4 to estimate the final kth largest score, that is $\hat{\alpha}$. According to $\hat{\alpha}$ and query function f, we can find the histogram that contains the kth largest record r. The center of the histogram is regarded as the kth largest record $\hat{\alpha}$. For each DG_i, we project $\hat{r}$ into dimension subset I_i (denoted by $\hat{r}|_{I_i}$). According to subfunction f_i, we can obtain ISC $f_i(\hat{r}|_{I_i})$. The value of k_i can be estimated as the number of records in "larger" area in dimension subset I_i. At last, we utilize methods in Section 4 to compute $Traveler\_TotalCost(k_i)$. We omit the details due to space limitation.

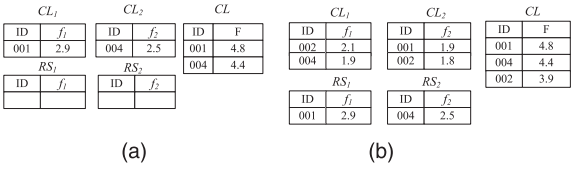


Fig. 7. Steps 1 and 2 in N-Way Traveler algorithm. (a) Step 1. (b) Step 2.

6 Maintenance

In many practical applications, there are frequent deletions and insertions in the record set. It is difficult for existing layer-based and view-based top-k algorithms, such as Onion, PREFER, and AppRI, to handle the problem. In Onion, if we delete a record r in the nth convex hull layer, all mth layers need to be recomputed, where m $\leq$ n. Computing convex hull is very expensive [4], [20]. In PREFER, in order to handle insertion and deletion, we need to reselect and rematerialize views. In AppRI, authors also point out that it is advisable to perform index maintenance in batches [5].[2] In this section, we propose efficient online DG maintenance algorithms, which are suitable for both DG and Extended DG. Update operation can be regard as "first deletion then insertion," therefore, we neglect the discussion about update operation.

As discussed in Section 3, DG index files and data files in each layer are stored separately. First, we discuss how to update DG index structure to handle insertion and deletion efficiently in Section 6.1. As discussed in Section 3, we adopt cluster-based schema to store original records in data files. In Section 6.2, we discuss how to handle insertion and deletion in data files.

6.1 DG Index Maintenance

6.1.1 Insertion

Algorithm 5 shows the insertion algorithm of DG. When inserting some record r, we need to locate the level in DG where r should be inserted (Lines 1-6). If r is not dominated by any record in the first layer of DG, r will be inserted into the first layer (Lines 1-2). Otherwise, if r is dominated by some records P_i in the first layer, we can find the longest path L from P_i to some record using Depth First Search (DFS) from P_i and each record in the path L dominates r (Lines 3-6). Assume that the length of path L is n. The longest path L guarantees that r cannot be dominated by any record in the $(n+1)$th layer. Thus, r should be inserted into the $(n+1)$th layer (Line 7). If r dominates several records C_i in the $(n+1)$th layer, all descendant records of C_i including C_i will be affected, and they are degraded into the next layer (Lines 8-14). All affected records, which can be found using DFS search from records C_i, are collected to form the set S (Line 9). Each record O in S is degraded into the next layer (Line 11). We build "parent-children" relationship between O and records in its current next layer. Remove the edges from records out of S to record O in S (Lines 13-14). Then, build the "parent-children" relationship between records in

2. Two temporal solutions for online maintenance of AppRI are suggested in [5]. For example, we can mark some deleted records. However, using temporal solutions, the quality of AppRI index will degrade with frequent updates. Authors in [5] point out that AppRI needs to be rebuilt periodically.

the nth layer with r (Line 15), and r with records in the $(n+2)$th layer (Line 16).

Algorithm 5 Insert Algorithm

Require: Input: DG: the dominate graph of the database; r: a record to be inserted.
1: **if** r is not dominated by any record at 1st layer of DG **then**
2: Set n=0
3: **else**
4: All record P_i at 1st layer of DG that dominate r are collected to form the set P.
5: Do DFS search from each P_i to find the longest the path L, and each record in L dominates r.
6: Set $n = |L|$
7: Insert r into the (n+1)th layer of DG.
8: **if** r dominates some records C_i in the (n+1)th layer of DG **then**
9: all descendant records of C_i(including C_i) are collected to form the set S.
10: **for** each record O in S **do**
11: O is degraded into its next layer
12: Build "parent-children" relationship between O and records in its current next layer.
13: **if** O has some other parent A that is not in set S **then**
14: Delete the directed edge from A to O.
15: Build the "parent-children" relationships between records in nth layer and r.
16: Build the "parent-children" relationships between records in r and $n+2$th layer.
17: report the updated DG.

Algorithm 6 Delete Algorithm

Require: Input: DG: the dominate graph of the database; r: a record to be deleted.
1: Find the position of r.
2: Delete r from nth layer of DG.
3: **for** each child C_i of r **do**
4: **if** C_i has no another parent in nth layer. **then**
5: Insert C_i into nth layer.
6: Build the "parent-children" relationships between records at $(n-1)$th layer and C_i.
7: Do $Del(C_i, n+1)$

Del(O,m)
1: Delete O from the mth layer of DG
2: **for** each child C_i of O **do**
3: **if** C_i has no another parent in mth layer **then**
4: Insert C_i into mth layer
5: Build the "parent-children" relationships between records in (m-1)th layer and C_i.
6: $Del(C_i, m+1)$.

6.1.2 Deletion

When we delete some record r in the nth layer, the simplest solution is that we can mark the record r as a *pseudorecord*. In this way, Advantage *Traveller* algorithm can still find the correct top-k answers. We discuss another deletion algorithm of DG, shown in Algorithm 6. When we delete a record r, we first find the position of r in DG (Line 1). Then, delete it from the nth layer (Line 2). For each child C_i of r, if and only if C_i has no other parents, C_i is promoted to the nth layer from the $(n+1)$ layer (Lines 4-5). Then, we build edges between records in the $(n-1)$th layer and C_i (Line 6). Obviously, promoting C_i from the $(n+1)$th layer to the nth layer means deleting C_i from the $(n+1)$th layer. Therefore, it will lead to the chain reaction in the remaining layers (Line 7).

6.2 Data File Maintenance

As discussed in Section 3, different storage schemas for data files will lead to different I/O costs. Obviously, insertion and deletion will affect the original storage schema. In order to handle updates, we discuss "Insertion," "Deletion," "Split," and "Merge" for data files.

As we know, for each layer in DG index, records are stored into different disk pages. We have two parameters Θ and θ ($\Theta > 2 \times \theta$), where Θ and θ denote the maximal and minimal number of records in each disk page. If the number of records in some disk page is larger than Θ, we need to split the disk page into two pages. If the number of records in some disk page is smaller than θ, we need to merge this page to another one.

6.2.1 Insertion

Definition 12. *Given a disk page P, r' is a record to be inserted into P. We define extension ratio Γ as follows:*

$$\Gamma(P) = |Parent(P) \cup Parent(r')| - |Parent(P)|,$$

where $\mathrm{Parent}(P) = \bigcup_{r \in P} \mathrm{Parent}(r)$ *and* $\mathrm{Parent}(r)$ *denotes the set of record r's parents.*

Assume that one record r should be inserted into some layer L. There are more than one disk page in this layer. We find the disk page P where $\Gamma(P)$ is minimal among all disk pages. Then, we insert record r into disk page P. After insertion, if there are larger than Θ records in P, it will lead to split operation, as discussed later.

6.2.2 Deletion

It is straightforward to delete record r from disk page P. After deletion, if there are less than θ records in P, it will lead to merge operation, as discussed later.

6.2.3 Split

If there are more than Θ records in disk page P, we need to split page P into two disk pages P_1 and P_2. According to record distance (defined in Definition 7), we group records in P into two clusters by K-Means algorithm. Each cluster corresponds to one disk page. Notice that we require each cluster has more than θ records.

6.2.4 Merge

If there are less than θ records in disk page P in some layer L, we need to merge P into another page in the same layer. If there is only one disk page in layer L, we ignore mergence operation. Otherwise, according to group distance (defined in Definition 8), we merge P into the nearest disk page P'. After mergence, if there are more than Θ records in the merged page, it will lead to split operation.

7 Experiments

In this section, we evaluate our methods in both synthetic and real data sets, and compare them with some classical top-k algorithms, such as TA [1], CA [1], Onion [4], PREFER [6], and some state art of top-k algorithms, such as AppRI [5], RankCube [21]. Since most top-k algorithms can only support linear query function, to enable fair performance comparison with these algorithms, we only use linear function in comparison study. Since our algorithm can support any monotone aggregate function, thus, we also test our algorithm under nonlinear function.

7.1 Experiment Setting

We implement TA, CA, and Onion[3] by ourself. We download the software PREFER from http://db.ucsd.edu/prefer/. The AppRI software is provided by authors

3. In order to find convex hull layers in Onion algorithm, we use the software Qhull [20] downloaded from http://www.qhull.org/.

TABLE 2
Layer Number in Different Distribution and Dimension

Dimension Number	Uniform	Gaussian	Correlated
2	401	334	2000
3	254	296	472
4	74	110	148
5	38	44	75

in [5]. Since RankCube [21] supports both selection condition and ranking condition, in order to enable performance comparison, we ignore the selection condition in RankCube [21]. After discussion with authors in [21], we implement RankCube as follows by ourself: first partition the data set into blocks according to Section 3.1.2 in [21], and then answer top-k query according to the query algorithm in Section 3.2.2 in [21]. All experiments are implemented by standard C++ and conducted on a P4 1.7 GHz machine of 1G RAM running Windows XP.

7.2 Data Sets

We use both synthetic and real data sets in our experiments. There are three kinds of synthetic data sets, i.e., Uniform, Gaussian, and Correlated data sets, which are denoted by U_i, G_i, and r_i, respectively. The cardinality of each data set is 1,000K. For uniform data sets, attribute values are "uniformly distributed" between 0 and 1. For Gaussian data sets, mean equals to 0.5 and variance equals to 1. We generate the correlated data set by the similar method in [7], specifically, we first generate a data set with uniform distribution in the dimension x_1; then, for each value v in the dimension x_1, we generate values in other $m-1$ dimensions by sampling a Gaussian distribution with mean v and fixed variance. In this way, records in the correlated data set are located close to the line from $(0,\ldots 0)$ to $(1,\ldots 1)$. The real data set, *server*, is KDD Cup 1999 data containing the statistics of 500K network connections (http://kdd.ics.uci.edu/databases/kddcup99/). We extract three numerical attributes (with cardinalities 569, 1,855, and 256) to create a three-dimensional data set: *count*, *srv-count*, *dest-host-count*. Another real data set, *house*, consists of 127k six-dimensional records, each of which represents the percentage of an American family's annual income spent on six types of expenditure: gas, electricity, water, heating, insurance, and property tax (http://www.ipums.org/).

7.3 Experiment 1 (DG Indexes)

We show the number of layers about DG in 2. From Table 2, there are two important findings: 1) The number of layers of DG in a record set with the high dimension is larger than that in the small dimension. As we know, the cardinality of skylines in the high dimension is larger than that in the small dimension. It means that the number of records in each layer of DG in the high dimension is larger than that in the small dimension. Therefore, the number of layers of DG in high dimension is smaller than that in the small dimension. 2) In the same dimension, the number of layers of DG in correlated record set is the largest among three distributions. Thus, the number of "domination relationships" in correlated data sets are larger than that in other distributions. The number of records in each layers in correlated data sets is the smallest among three distributions.

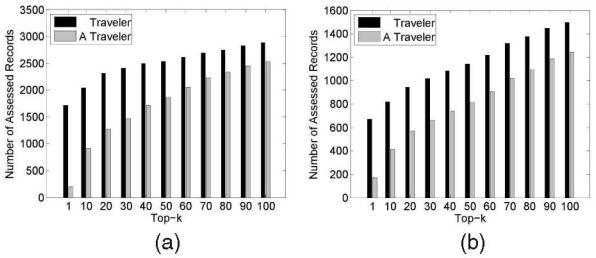


Fig. 8. The number of assessed records during Top-K query. (a) G_5, $|D| = 1M$, $|m| = 5$. (b) r_5, $|D| = 1M$, $|m| = 5$.

7.4 Experiment 2 (Compare Traveler and Advanced Traveler)

We evaluate the optimization technique, that is pseudorecord, in the experiment. We use the number of records that are assessed as the measure for *Traveler* algorithm. In Advanced *Traveler* algorithm, we introduce pseudorecords to reduce the cost. Due to space limit, only a subset of our experiments are present here.

Observed from Figs. 8a and 8b, in five-dimensional data sets, Gaussian (G_5) and Correlated (r_5), pseudorecords lead to the great reduction in the cost.[4] When evaluating the cost, assessed pseudorecords also count. Even though we count assessed pseudorecords, Fig. 8 shows that the number of all assessed records in Advanced *Traveler* are much less than that in basic *Traveler*, which confirms the efficiency of the pseudorecord technique.

7.5 Experiment 3 (Evaluating Cluster-Based Storage)

In order to estimate I/O cost, we do the following simulation experiments. According to the storage framework shown in Fig. 3, each data file corresponds to each layer. Assume that each disk page can store 10 records. All records in a data file are clustered into different groups according to the cluster method in Section 3, and each group corresponds to one disk page. When some record r in disk page P needs to be assessed and computed, all other records r' in the same disk page P are also retrieved and cached in the memory. For simplicity of analysis, we assume that all cached records will not be swapped out of memory when performing *Traveler* algorithm. From Fig. 9, we know that there are at most 3,000 disk pages cached in memory, which consume only 12 Mbytes of the memory. Therefore, the above assumption is reasonable for a PC machine. The number of I/O times is evaluated in Fig. 9. Observed from Fig. 9, the I/O times in cluster-based method is only about $\frac{1}{10}$ of that in noncluster approach, which confirms that our cluster-based storage is efficient to reduce I/O cost.

7.6 Experiment 4 (Comparison with Existing Top-K Algorithms)

In the experiment, we compare our methods with existing algorithms. First, the performance of Advanced *Traveler* is compared with two representative layer-based top-k algorithms, such as *Onion* [4] and *AppRI* [5]. In the offline phase, observed from Figs. 10a and 10b, we need the least

4. *Traveler* is denoted by "*Traveler*" in Fig. 8. Advanced *Traveler* is denoted by "A *Traveler*" in Figs. 8, 10, and 12.

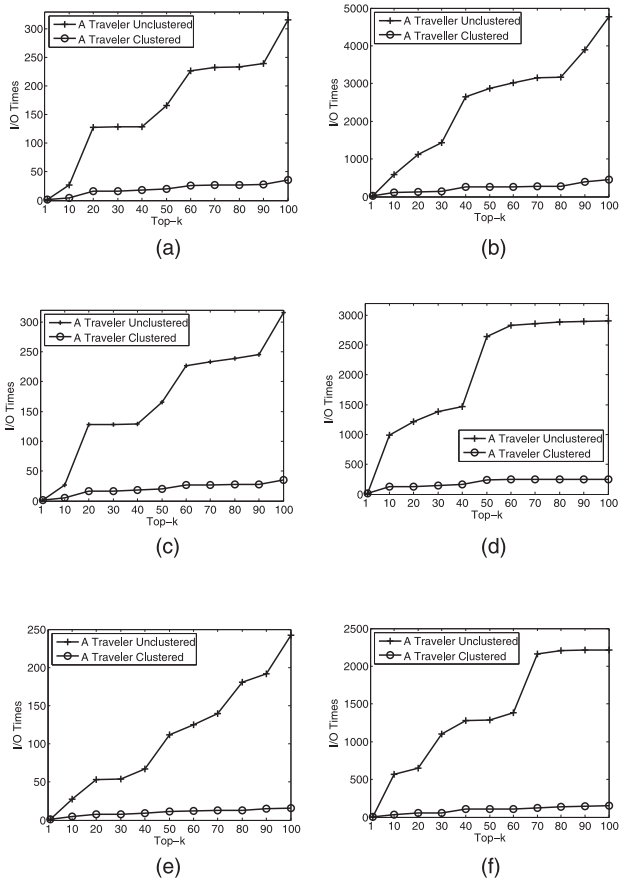


Fig. 9. I/O cost during top-K query. (a) I/O Times in U_3, $|D| = 1\text{M}$, $|m| = 3$. (b) I/O times in U_5, $|D| = 1\text{M}$, $|m| = 5$. (c) I/O times in G_3, $|D| = 1\text{M}$, $|m| = 3$. (d) I/O times in G_5, $|D| = 1\text{M}$, $|m| = 5$. (e) I/O times in r_3, $|D| = 1\text{M}$, $|m| = 3$. (f) I/O times in r_5, $|D| = 1\text{M}$, $|m| = 5$.

construction time to build DG index. In the online phase, as shown in Figs. 10c and 10d, the number of assessed records in Advanced *Traveler* algorithm is much smaller than that in *AppRI* and *Onion*. The query response time in Figs. 11a and 11b also confirm the performance of *Traveler*. Furthermore, we also report I/O times in different layer-based algorithm. Although existing layer-based algorithms perform sequential access rather than random access in Advanced *Traveler* algorithm, the number of I/O times is much larger than that in Advanced *Traveler* algorithm. As a consequence, the I/O cost of our algorithm is still smaller than that of the existing layer-based algorithms.

We also compare Advanced *Traveler* with some nonlayer based top-k algorithms, that are TA, CA, PREFER, and *RankCube*.[5] In CA, we only count the number of random access times. Observed from Fig. 12, Advanced *Traveler* outperforms other algorithms in both the number of assessed records and query response time.

In experiments, we also compare *Traveler* with other methods in five-dimensional data sets. The results also confirm that our algorithm has a significant improvement. Due to space limit, we do not present here. For higher dimension (>5), we report our performance in Experiment 4. Furthermore, we observed that the increasing trend of the number of assessed records in *Traveler* is much slower than

5. RankCube is denoted by "RCube" in Fig. 12.

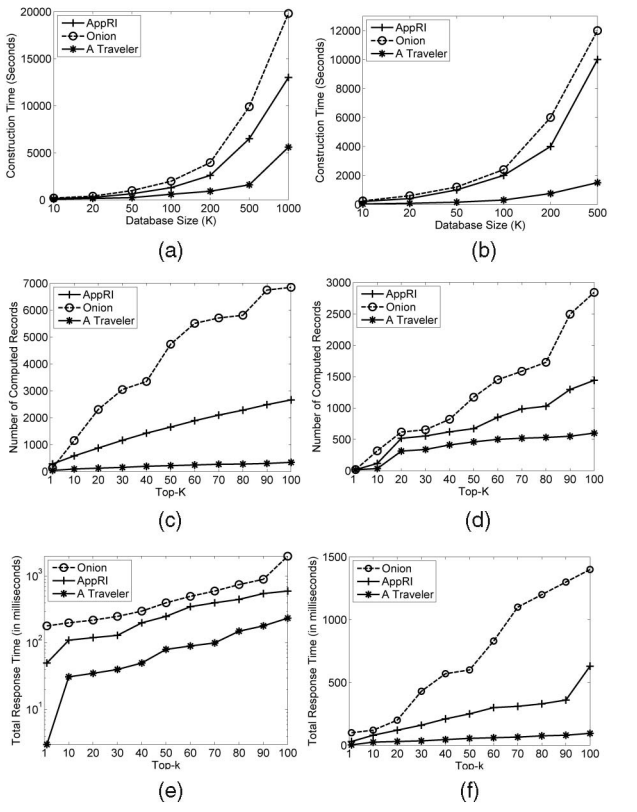


Fig. 10. Comparison with layer-based algorithms. (a) Construction time in U_3. (b) Construction time in real data set. (c) Number of assessed records in U_3, $|D| = 1M$, $|m| = 3$. (d) Number of assessed records in server data set, $|D| = 500K$, $|m| = 3$. (e) Query response time in U_3, $|D| = 1M$, $|m| = 3$. (f) Query response time in server data set, $|D| = 500K$, $|m| = 3$.

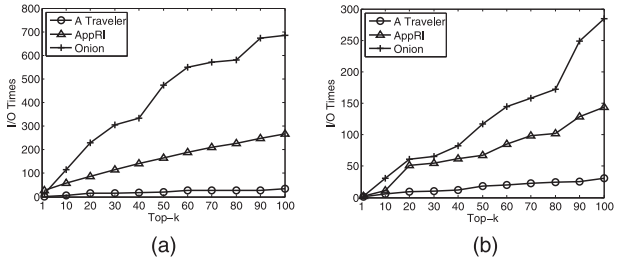


Fig. 11. I/O times in layer-based algorithms. (a) I/O times in U_3, $|D| = 1M$, $|m| = 3$. (b) I/O times in server data set, $|D| = 500K$, $|m| = 3$.

that in other methods in Figs. 10c, 10d, 12a, and 12b. That can be explained by Theorem 3. Given a query function F, $\overline{S_2}$ in top-1 and top-100 are a little different from each other. Therefore, there is a small difference between $|Skyline(\overline{S_2})|$ in top-1 query and top-100 query of Advanced Traveler algorithm.

7.7 Experiment 5 (Nonlinear Query Function)

Most existing top-k algorithms (such as Onion, Prefer, RCube, and AppRI) cannot work when query function is a nonlinear function. Therefore, we only compare our Traveler algorithm with TA and CA algorithm in this set of experiments. We define query function as $F = \sum_{i=1}^{i=m} a_i * x_i^2$.

Due to space limit, we only show experiment results on two real data sets, that are *server* and *house* data sets. Observed from Fig. 13, Traveler algorithm outperforms TA

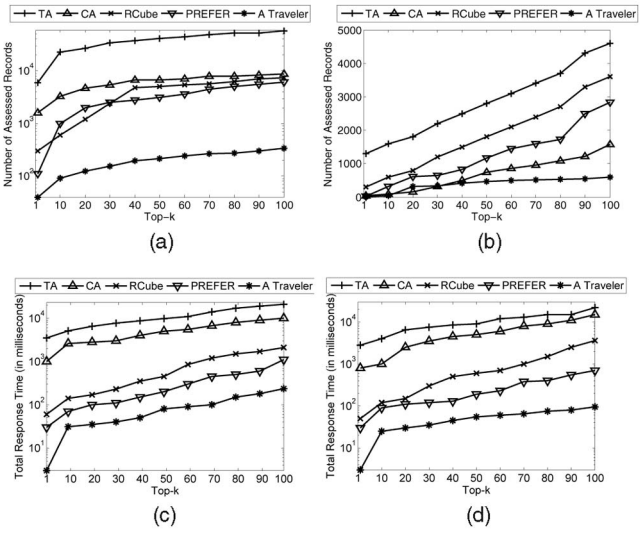


Fig. 12. Comparison with nonlayer-based algorithms. (a) Number of assessed records in U_3, $|D| = 1M$, $|m| = 3$. (b) Number of assessed records in real data set, $|D| = 500K$, $|m| = 3$. (c) Query response time in U_3, $|D| = 1M$, $|m| = 3$. (d) Query response time in real data set, $|D| = 500K$, $|m| = 3$.

and CA algorithm by order of magnitude. The results on other synthetic data sets are similar with that in these two real data sets.

7.8 Experiment 6 (DG Maintenance)

Our algorithm is a layer-based method. We need to update DG to handle the insertion and deletion in the record set. In the experiment, we evaluate the DG maintenance algorithms proposed in Section 6 and compare them with other layer-based algorithms. In Section 6, we show that DG maintenance algorithms have time complexity $O(|D|^2)$ in the worst case. In fact, the practical performance of DG maintenance is quite desirable. For insertion, for each data set, we generate another data set with 10K records and the same distribution. For deletion, we randomly delete 10K records from each data set. There are 1,000K records in each original data set (U_3, G_3, and r_3). We report the running time in Fig. 14. Observed from Fig. 14, DG maintenance algorithm is suitable in the online process to handle insertion and deletion operations, since the running times are less than 140 and 500 seconds for 10K insertions and deletions, respectively. Furthermore, we find that the cost in R_3 is larger than U_3 in Fig. 14. The reason is that insertion and deletion may affect more layers in correlation data sets than uniform data sets.

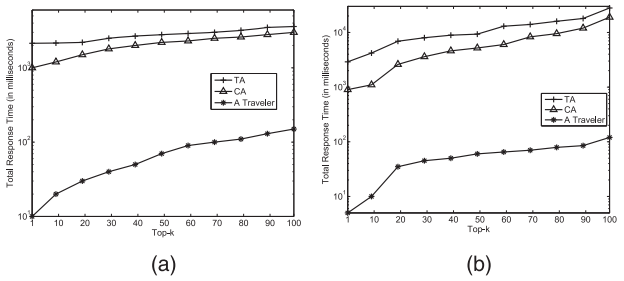


Fig. 13. Performance on nonlinear function. (a) Total response time in household data set. (b) Total response time in server data set.

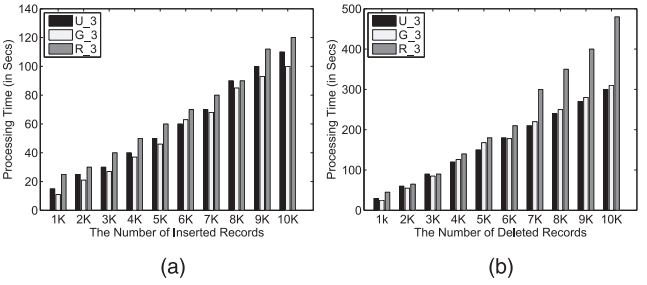


Fig. 14. Evaluating DG maintenance. (a) $Insertion$. (b) $Deletion$.

Fig. 14 does not show the maintenance cost in other layer-based algorithms, such as Onion and AppRI. As discussed in Section 6, these algorithms need to reconstruct layers to handle insertion and deletion. For example, in order to handle the same 10K insertions in an 1,000K data set, we need about 19,000 and 13,000 seconds to reconstruct layers in Onion and AppRI, respectively. Therefore, our DG has the "light" maintenance cost.

7.9 Experiment 7 (High Dimension and Worst-Case Testing)

In the experiment, we first test N-Way *Traveler* algorithm in 10-dimensional data set. Most top-k algorithms are evaluated on the data set with no more than five dimensions, such as Onion, PREFER, AppRI, and RankCube. For example, in Onion, it is very difficult to find convex hull in a large data set with 10 dimensions. Therefore, we only use TA and CA algorithms here for comparison with N-Way *Traveler*. In N-Way *Traveler*, we build two DGs for each five dimensions, respectively. Observed from Figs. 15a and 15b, we find that N-Way *Traveler* outperforms TA and CA by orders of magnitude in both the number of assessed records and total response time. We also report the number of accessed layers in each DG_i in Fig. 15c. In our method, the worst case happens when all records are skyline points. In order to test Advanced *Traveler* algorithms in the special case, we generate a five-dimensional data set, where all records are skyline points. Figs. 16a and 16b show that, due to pseudorecord optimization technique, Advanced *Traveler* still works well in the worst case.

7.10 Experiment 8 (Testing Cost Estimation)

In this set of experiments, we evaluate the estimation method in Section 7.10. We adopt the equiwidth histogram, namely, we partitions the whole data space into c^d equal-size buckets, where c is the histogram resolution, and d is the dimensionality. In experiments, we set the histogram resolution $c = 20$. In order to estimate $Skyline(\overline{S_2})$, we adopt the sampling technique. Specifically, we randomly choose 1K records from the original database as sampling points. According to the estimated ISC, we remove the points whose scores are larger than the estimated ISC. After that, we use the left sampling points to estimate parameters A and B (see Section 7.10). Then, we can use $Alog^B|\overline{S_2}|$ to estimate $|Skyline(\overline{S_2})|$. At last, the total estimated cost can be evaluated by $k - 1 + Alog^B|\overline{S_2}|$. We measure the precision of the estimation method as the average estimation error for all the queries in a workload. For each query, the actual and estimated number of assessing records are denoted as act_i and est_i. The estimation error is measured

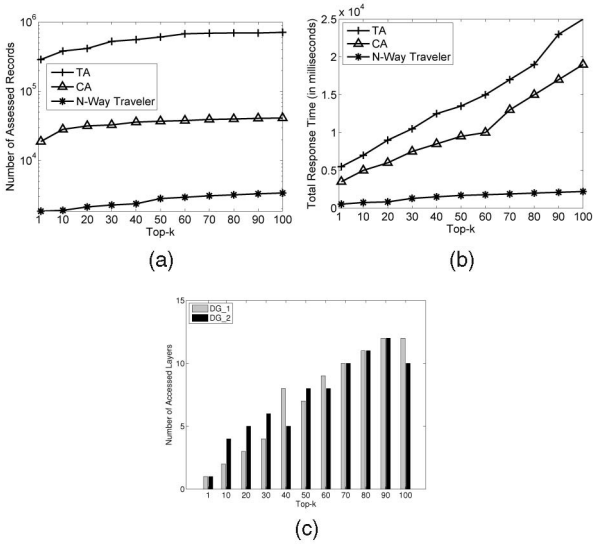


Fig. 15. Evaluating Traveler in high dimension. (a) Number of assessed records, $|m| = 10$, $|D| = 1\text{M}$. (b) Query response time, $|m| = 10$, $|D| = 1\text{M}$. (c) Number of accessed layers, $|m| = 10$, $|D| = 1\text{M}$.

as $|est_i - act_i|/act_i$. In experiments, we report the average error for 100 queries in Fig. 17. Observed from Fig. 17a, the maximal estimation error ratio is smaller than 0.5. Furthermore, the methods in Section 7.10 can provide more accurate results in uniform data set. As shown in Fig. 17b, with the increase of dimensions, the precision of the estimation methods grows down.

8 Related Work

Top-k preference queries (top-k queries for short) are very popular in many real applications, such as image databases [22], sensor networks [23], and web search [5]. As a consequence, many algorithms for efficiently answering top-k queries have been proposed. The first category is based on *ranked list* [2], [3], [1]. The threshold algorithm (TA) algorithm constitutes the state of the art for top-k query. The methods in this category sort the data in each dimension and aggregate the rank by parallel scanning each list sequentially (sequential access). For each record identifier encountered in the sequential access, it immediately accesses (random access) other lists for their scores. Since there are two kinds of accesses: sequential and random access. In [24], Bast et al. proposed an integrated view of the scheduling issues for these two kinds of accesses. Different from our problem, in some top-k algorithms, only sequential access is allowed, such as NRA proposed by Fagin et al. in [1] and LARA

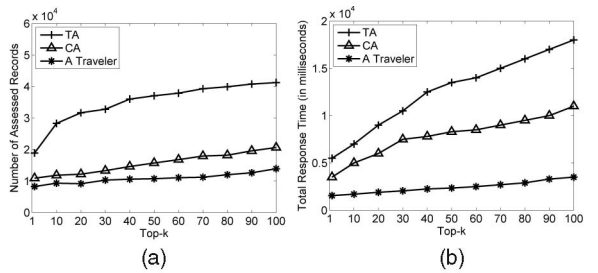


Fig. 16. Evaluating Traveler in the worst case. (a) Number of assessed records in the worst case, $|D| = 100\text{K}$, $|m| = 5$. (b) Query response time in the worst case, $|D| = 100\text{K}$, $|m| = 5$.

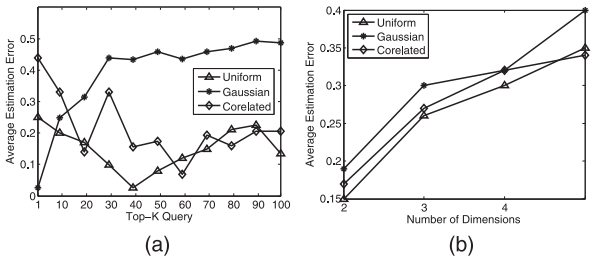


Fig. 17. Accuracy of cost estimation in Traveler algorithm. (a) Accuracy of cost estimation versus Top-K ($|m| = 3$, $|D| = 1{,}000\text{K}$). (b) Accuracy of cost estimation versus number of dimensions ($\text{K} = 100$, $|D| = 1{,}000\text{K}$).

proposed by Mamoulis et al. in [25]. These algorithms find applications, such as data streams. In these applications, it is impossible or very expensive to perform random access. The second category is *layer-based*, such as Onion [4] and AppRI [5]. In Onion, proposed by Chang et al., given a linear query function, records of interest can only lie in the convex hull. Thus, the Onion technique in a preprocessing step computes the convex hull of the record space. Storing all records of the first hull in a file and proceeds iteratively computing the convex hulls of the remaining records until no records are left. In AppRI, proposed by Xin et al, a record t is put in the layer l iff 1) for any possible linear queries, t is not in top l-1 results; and 2) there exists one query such that t belongs to top l results. Different from our method, in both Onion and AppRI, when the algorithm accesses to the nth layer, all records before the nth (including the nth layer) layer need to be accessed and computed by query function. The third category is *view*-based, such as PREFER [6] and LPTA [7]. PREFER, proposed by Hristidis et al. [6], uses a view sequence R_v, which ranks the records of a relation R according to a view preference vector $\overrightarrow{v}$. In order to answer a top-k query Q with preference vector $\overrightarrow{q}$, we need to compute the water mark in the view sequence R_v that guarantees that we can obtain the first answer in the query Q. Repeat the above the process until we get the top-k answers. In [7], Das et al. proposed another view-based algorithm, LPTA, which maintains some sorted record ID lists according to the view's preference functions. Similar with TA algorithm, traversing record ID lists until the top-k answers are retrieved. Parallel to top-k queries, there is another type of queries, *skylines*. The skyline operator was first introduced by Borzonyi et al. in [8]. The existing skyline algorithms can be categorized into

1. block nested loops (BNL) [8],
2. divide-and-conquer (D & C) [8],
3. bitmap and index [10],
4. nearest neighbor (NN) [11], and
5. branch-and-bound Skyline (BBS) [9].

9 Conclusions

In this paper, based on the intrinsic connection between top-k problem and dominant relationships, we propose an efficient indexing structure DG and a top-k query algorithm *Traveler*, which can support any monotone aggregate function. Most importantly, we propose an analytic cost model for our basic query algorithm and prove that the query cost is directly related to the cardinality of skylines in the record set. Furthermore, we extend basic *Traveler* to its advanced version by introducing *pseudorecords* to reduce the cost, and design N-way *Traveler* algorithm to handle the

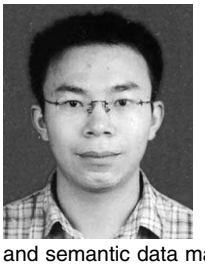
high-dimension problem. Extensive experiments confirm that our approaches have significant improvement over the previous methods. Furthermore, due to advantage of DG indexing structure, we propose the "light" DG maintenance algorithm to handle insertion and deletion in the online process. Since our DG index stores all dominant relationships in the original data set, thus, we can make use of DG index to support dominant relationship analysis. In our future work, we will study how to apply DG index in dominant relationship analysis.

Acknowledgments

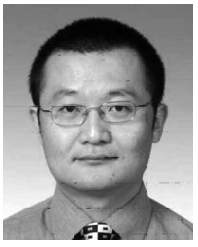
Lei Chen is supported by Hong Kong RGC NSFC JOINT Grant under Project No. N_HKUST612/09 and the NSFC Projects under grant No. 60736013, 60933011, 60873022, and 60903053. Lei Zou is supported by the NSFC under Grant No.61003009 and the Research Fund for the Doctoral Program of Chinese Higher Education No. 20100001120029 and Beijing Science & Technology Program Z101101005010003. This is an extended version of the paper "Dominant Graph: An Efficient Indexing Structure to Answer Top-K Queries" that was presented in the *Proceedings of 24th International Conference on Data Engineering (ICDE)*, pp. 536-545, 2008.

References

[1] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," *Proc. Symp. Principles of Database Systems (PODS),* 2001.
[2] S. Nepal and M.V. Ramakrishna, "Query Processing Issues in Image (Multimedia) Databases," *Proc. 15th Int'l Conf. Data Eng. (ICDE),* 1999.
[3] U. Güntzer, W.-T. Balke, and W. Kießling, "Optimizing Multi-Feature Queries for Image Databases," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2000.
[4] Y.-C. Chang, L.D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J.R. Smith, "The Onion Technique: Indexing for Linear Optimization Queries," *Proc. ACM SIGMOD,* 2000.
[5] D. Xin, C. Chen, and J. Han, "Towards Robust Indexing for Ranked Queries," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2006.
[6] V. Hristidis, N. Koudas, and Y. Papakonstantinou, "Prefer: A System for the Efficient Execution of Multi-Parametric Ranked Queries," *Proc. ACM SIGMOD,* 2001.
[7] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis, "Answering Top-K Queries Using Views," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2006.
[8] S. Börzsönyi, D. Kossmann, and K. Stocker, "The Skyline Operator," *Proc. 17th Int'l Conf. Data Eng. (ICDE),* 2001.
[9] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An Optimal and Progressive Algorithm for Skyline Queries," *Proc. ACM SIGMOD,* 2003.
[10] K.-L. Tan, P.-K. Eng, and B.C. Ooi, "Efficient Progressive Skyline Computation," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2001.
[11] D. Kossmann, F. Ramsak, and S. Rost, "Shooting Stars in the Sky: An Online Algorithm for Skyline Queries," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2002.
[12] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms.* MIT Press, 2001.
[13] D. Campbell and R. Nagahisa, "A Foundation for Pareto Aggregation," *J. Economical Theory,* vol. 64, pp. 277-285, 1994.
[14] M. Voorneveld, "Characterization of Pareto Dominance," *Operations Research Letters,* vol. 32, no. 3, pp. 7-11, 2003.
[15] C. Li, B.C. Ooi, A.K.H. Tung, and S. Wang, "DADA: A Data Cube for Dominant Relationship Analysis," *Proc. ACM SIGMOD,* 2006.
[16] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou, "Branch-and-Bound Processing of Ranked Queries," *Information Systems,* vol. 32, no. 3, pp. 424-445, 2007.
[17] R.J. Lipton, J.F. Naughton, and D.A. Schneider, "Practical Selectivity Estimation through Adaptive Sampling," *Proc. ACM SIGMOD,* 1990.
[18] R.J. Lipton and J.F. Naughton, "Query Size Estimation by Adaptive Sampling," *Proc. Symp. Principles of Database Systems (PODS),* 1990.
[19] S. Chaudhuri, N.N. Dalvi, and R. Kaushik, "Robust Cardinality and Cost Estimation for Skyline Operator," *Proc. 22nd Int'l Conf. Data Eng. (ICDE),* 2006.
[20] C.B. Barber, D.P. Dobkin, and H. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Trans. Math. Software,* vol. 22, pp. 469-483, 1996.
[21] D. Xin, J. Han, H. Cheng, and X. Li, "Answering Top-k Queries with Multi-Dimensional Selections: The Ranking Cube Approach," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2006.
[22] S. Nepal and M. Ramakrishna, "Query Processing Issues in Image(Multimedia) Databases," *Proc. 15th Int'l Conf. Data Eng. (ICDE),* 1999.
[23] M. Li and Y. Liu, "Iso-Map: Energy-Efficient Contour Mapping in Wireless Sensor Networks," *IEEE Trans. Knowledge and Data Eng.,* vol. 22, no. 5, pp. 699-710, May 2010.
[24] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum, "IO-Top-k: Index-Access Optimized Top-k Query Processing," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2006.
[25] N. Mamoulis, K.H. Cheng, M.L. Yiu, and D.W. Cheung, "Efficient Aggregation of Ranked Inputs," *Proc. 22nd Int'l Conf. Data Eng. (ICDE),* 2006.



Lei Zou received the BS and PhD degrees in computer science at Huazhong University of Science and Technology (HUST), China, in 2003 and 2009, respectively. He has been an assistant professor in the Institute of Computer Science and Technology of Peking University, China, since 2009. Before that, he visited Hong Kong University of Science and Technology, and University of Waterloo, Canada. His research interests include graph database, data mining, and semantic data management. He is a member of the IEEE.



Lei Chen received the BS degree in computer science and engineering from Tianjin University, China, in 1994, the MA degree from Asian Institute of Technology, Thailand, in 1997, and the PhD degree in computer science from the University of Waterloo, Canada, in 2005. He is now an assistant professor in the Department of Computer Science and Engineering at Hong Kong University of Science and Technology. His research interests include uncertain databases, graph databases, multimedia and time series databases, and sensor and peer-to-peer databases. He is a member of the IEEE.