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

Lei Zou [1] , Lei Chen [2]

[1]*Huazhong University of Science and Technology*
1037 Luoyu Road, Wuhan, P. R. China
[1]`zoulei@mail.hust.edu.cn`

[2]*Hong Kong University of Science and Technology*
Clear Water Bay, Kowloon, Hong Kong
[2]`leichen@cse.ust.hk`

***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 relationship between records, and based on which, we propose an efficient layer-based indexing structure, Dominant Graph (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 basic algorithm is directly related to the cardinality of skyline points in the record set (see Theorem 3.2). Based on the cost analysis, we propose the optimization technique, pseudo record, to 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. Finally, extensive experiments demonstrate that our proposed methods have significant improvement over its counterparts, including 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 [1], one of state art of top-k algorithms. Furthermore, our method can support any aggregate monotone query function.**

I. 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. Top-k queries are very popular in many applications. For example, a job seeker may want to find the best jobs fit to her preferences, such as near to her home, high salary, and short working time. For different applicants, they may have their own ranking by assigning different weights. Some may put a high weight on "salary", while others care more about "reputation" of the company. No matter what their preferences are, all these queries belong to top-k preference queries.

A. Motivation

Due to the popularity of top-k queries, there are many solutions that have been proposed in the database literature. Basically, 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[2], [3], [4].

2. Layer-based. The algorithms in this category organize all records into consecutive layers, such as Onion [5] and AppRI [1]. The organization strategy is based on the common property among the records, such as the same convex hull layer in Onion [5]. Any top-k query can be answered by up to k layers of records.

3. View-based. Materialized views are used to answer 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) all existing algorithms in these two categories only support linear query function, such as [5], [1], [6], [7]; 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, *skylines*, 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 efficiently, such as block nested loops [8], divide-and-conquer [8], bitmap [10], index [10], nearest neighbor [11] and branch and bound [9].

In fact, both top-k and skyline query are not independent of each other, they are connected by the dominant relationships. Thus, we would like to make use of dominant relationship to answer 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 top-k answer. Therefore, top-k answers can be derived from *offline* generated dominant relationships among all the records.

B. Our Contributions

In this paper, with the consideration of the intrinsic connection between top-k problem and dominant relationships, we design an efficient indexing structure, **D**ominant **G**raph (DG), to support top-k query. We derive the most important property in DG: *the necessary 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 3.1). Benefiting from the property, the search space (that is the number of retrieved records from the record set to answer top-k query) of 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 3.2). Based on cost analysis, we propose the optimization technique, *pseudo record*, to reduce the search space. Furthermore, *DG* index can support any aggregate monotone query function and has the "light" maintenance cost.

In this paper, we make the following contributions:

1) We represent the dominant relationships among the records into a *partial order* graph, called ***D**ominant **G**raph* (DG). In DG index, records are linked together according to the dominant relationship. Based on offline built DG, we propose *Traveler* algorithm, which transforms top-k query into a graph traversal problem.

2) We propose an analytic model to the query cost of Traveler. To improve the query performance, we introduce a novel concept, *pseudo records*, and extend *Traveler* to its advanced version, Advanced *Traveler* algorithm.

3) In order to handle high-dimensional datasets, we propose N-Way *Travel* algorithm for top-k query. We also design efficient DG maintenance algorithms for deletion and insertion operations.

4) 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 [1], one of state art of top-k algorithms. Due to the optimization technique, pseudo record, Advanced *Traveler* algorithm still works well in the worst case, that is no dominant relationship in the original record set.

The remainder of the paper is organized as follows. DG and basic *Traveler* algorithm for top-k query is proposed in Section II. The cost of the basic algorithm is discussed in details in Section III. In Section IV, we extend basic *Traveler* algorithm to its advanced version, and also propose the solution to handle high dimension . We discuss DG maintenance in Section V. We evaluate the efficiency of our DG approach with extensive experiments in Section VI. Related work is discussed in Section VII. Finally, we conclude the paper in Section VIII.

II. TOP-K QUERY BASED ON DG

For top-k query, the aggregate monotone query function F is defined as follows:

TABLE I
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	x_i	the ith dimension in record

Definition 2.1: **Aggregate Monotone Function [2].** An aggregation function F is a *monotone* if $F(x_1...x_n) \leq F(x'_1...x'_n)$, whenever $x_i \leq x'_i$ for every i.
Intuitionally, if we can pre-rank all data records using dominant relationships in the offline phase, we can answer a online top-k query by traversing the "pre-ranked list". Table I lists commonly used symbols in this paper.

A. Basic Traveler Algorithm

Definition 2.2: **Dominate [12].** Given two records R and R' in a multi-dimension space, we say that R *dominates* R' if and only if they satisfy the following two conditions: 1) in any dimension x_i, the value of R is larger than or equal to R', i.e. $R.x_i \geq R'.x_i$; 2) there must exist at least one dimension x_j, $R.x_j > R'.x_j$.

The query function in top-k preference query is an aggregate monotone function [2], [5], [6], [1], [7], if record R dominates another record R', the score of R is always larger than R'. Notice that, Definition 2.2 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 2.3: **Maximal Layers [12].** Given a set S of records in a multi-dimension 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 $S - \bigcup_{j=1...i-1} L_j$.

Definition 2.4: **Dominant Graph.** Given a set S of records in a multi-dimension space, S has n nonempty maximal layers L_i, i=1 ... n. The records R in ith maximal layer and records R' in (i+1)th layer form a bipartite graph g_i, i=1...(n-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 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 in the offline phase, we can use any skyline algorithm to find each layer of DG. Then we build the "parent-children relationship" between the ith layer and the *(i+1)*th layer.

Notice that, DG is stored independently as the indexing structure for the record set. A DG index is shown in the following example.

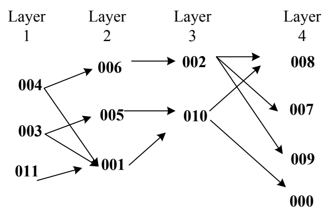
Example 1 (Running Example) We have a record set D, as shown in Fig. 1a. The DG of the record set is shown in Fig.

1b. Given a query function F = 0.6 * X + 0.4 * Y, we want to obtain top-2 answers.

Top-2 Query:
F= 0.6 * x + 0.4 * y

TID	X	Y
000	1	563
001	193	808
002	585	479
003	350	895
004	822	809
005	174	858
006	710	513
007	303	14
008	91	364
009	147	165
010	100	800
011	351	810

(a) A Database *D*



(b) DG of the database *D*

Fig. 1. Running Example

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 2.1: Given a top-k query function F over record set D, if *R* $\in D$, 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 2.4, 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-(k-1) answers. □

Based on Lemma 2.1, we can convert a top-k query into a graph traversal problem in DG. The Basic *Traveler* algorithm is presented in Algorithm 1.

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 $RS$
```

In Basic *Traveler*, firstly, we compute scores for all records in the 1st layer of DG according to the query function F and insert them into the max-heap Candidate List (i.e. *CL*) in non-ascending 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 Line 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 $(n-1)$th iteration step. In Line 5-9, for each child C of record R, if C has another parent that is not in RS now, C will not be accessed 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 (Line 10-11). The record with the largest score in CL now must be the (n+1)th largest answer, which is moved from CL to RS (Line 12). The above process (Line 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 1st layer of DG, they are computed by query function. They are inserted into CL, as shown in Fig. 2a. 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 removed 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. 2b. Now, top-2 answers have been found in RS. The algorithm terminates.

CL

ID	Score
004	816.8
003	568
~~011~~	~~534.6~~

RS

ID	Score
004	816.8

(a) Step 1

CL

ID	Score
006	631.2
~~003~~	~~568~~

RS

ID	Score
004	816.8
006	631.2

(b) Step 2

Fig. 2. Step 1 and 2 in the Running Example

B. Correctness of Basic Traveler

Theorem 2.1: 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 (a>1), we must find a record R in the 1st 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 1st 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 *n*th 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 2.1, it also indicates that *C* cannot be in top-(n+1). Therefore, only when all *C*'s parents are in the *RS* in *n*th iteration step, *C* has the chance to be in top-(n+1) answers, that is, *C* will be computed and inserted into *CL*

(Lines 7-12). Therefore, we cant obtain the correct *(n+1)*th largest answer in the *n*th 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, according to induction method, we can find the correct top-k answers in Basic *Traveler* algorithm. □

III. Cost Analysis

In this section, we analyze the cost of Basic Traveler algorithm, which is defined as follows.

Definition 3.1: The cost of Basic Traveler algorithm is the number of records to be accessed and computed by query function during top-k query.

Before the formal cost analysis, we present the following Property 3.1 and Lemma 3.1, which are used in the proof of Theorem 3.1.

Property 3.1: 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. (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 2.1, 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 3.1: A record *R* in the *a*th (a>1) layer needs to be computed in Travel 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=$\{R_i|R_i$ is computed by query function in Basic $Traveler$ $\}$. S_1 denotes the total search space in the algorithm. Obviously, according to Definition 3.1, 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=$\{R_i|R_i$ is in the final top-(k-1) 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. 3, 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 3.1: S_1=$S_2 \cup S_3$.

Proof.(*proof by contradiction*).

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

$$R' \notin (S_2 \cup S_3) \rightarrow (R' \in \overline{S_2}) \cap (R' \in \overline{S_3})$$

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 3.1, R' cannot be computed by query function in $Travel$ algorithm, namely, $R' \notin S_1$, which is contradicted to the assumption.

2) On the other hand, 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 1st layer or in ath (a>1) layer. If R' is in 1st 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 3.1, 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), we know S_1=$S_2 \cup S_3$. □

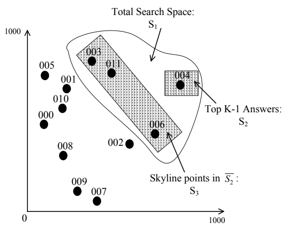


Fig. 3. Search Space in Traveler

Theorem 3.2: Given a record set D and a top-k query, the cost of Basic Traveler algorithm can be evaluated as follows:

$$Cost = k - 1 + |skyline(\overline{S_2})| \approx k + |skyline(D)|,$$

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

Proof. According to Theorem 3.1, it is clear to know $Cost$=$|S_1|$ = $|S_2|$+$|S_3|$= k-1+$|skyline(\overline{S_2})|$. Since, in top-k query, $k \ll logn$, where n is the number of records in D. Therefore, $\overline{S_2}$=D-$S_2$$\approx D$. $\overline{S_2}$ has the similar underlying data distribution with the original data set D. Therefore, it is straightforward to know Theorem 3.2 holds. □

To estimate the cardinality of skylines ($|skyline(D)|$) in D, some work have been proposed, such as [13], [14]. For example, if we know the underlying data distribution for D, $|Skyline(D)|$ can be evaluated by $n * \int_{[0,1]^d} f(\overline{x})(1 - F(\overline{x}))^{n-1} d\overline{x}$, where $\overline{x} = (x_1, ..., x_d)$ is d-dimension variable, $f(\overline{x})$ denotes the joint density function on $\overline{x}$, $F(\overline{x})$ denotes the joint distribution function on $\overline{x}$.

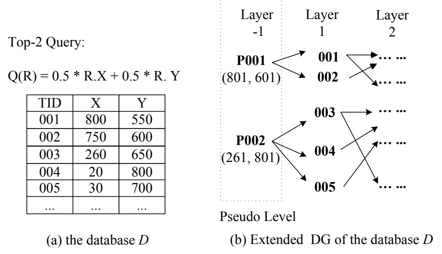
Obviously, with the increase of dimension, $|skyline(D)|$ gets larger. We will address high dimensionality problem in Section IV-C by proposing an N-Way *Traveler* algorithm.

IV. Advanced Traveler Algorithm

A. Pseudo Records

According to Line 1 in Basic *Traveler* algorithm, we need to compute all records in the 1st layer of DG. In order to reduce cost in the 1st layer, we introduce *pseudo records* to prune as many false candidates as possible. Given another database *D* and a top-2 query in Fig. 4a, different from the running example in Fig. 1, record 001-005 are all in the first layer of DG. We introduce some pseudo records to dominate records in the 1-st layer, and build an *Extended Dominate Graph* (Extended DG), as shown in Fig. 4b. According to similar traverse method with Basic *Traveler* algorithm, we obtain the top-2 answers only accessing record P002, P001 and 001 and 002, where record P001 and P002 are introduced pseudo records. Thus, after introducing pseudo records, the cost is 4, smaller than 5 in Basic *Traveler*. Observed from the above motivation example in Fig. 4, we know that pseudo records are useful to reduce the search space. We would point out that it is only necessary to introduce pseudo records when $L_1.size$ is large, where $L_1.size$ is the number of records in the 1st layer. In practice, we set the threshold $\theta = \frac{|page|}{|record|}$, where $|page|$ and $|record|$ are the number of bytes of a disk page and a record respectively. When $L_1.size > \theta$, we introduce pseudo records by the following method.

We cluster all records in the first layer into different clusters by K-Means algorithm according to Euclidean distance [15]. For each cluster, we introduce a pseudo record *P* (i.e. parent) dominating all records R_i in the cluster. After that, we remove some pseudo records that are dominated by other introduced pseudo records, which means that there are no dominant relationship among these introduced records. These introduced records form the layer L_{-1}. We build the "parent-children relationship" between the pseudo records in L_{-1} and the records in the 1st layer. If $L_{-1}.size > \theta$, iteratively, we form *(-n)*th layer by the same method until $L_{-n}.size < \theta$.



TID	X	Y
001	800	550
002	750	600
003	260	650
004	20	800
005	30	700
...	...	...

(a) the database *D* (b) Extended DG of the database *D*

Fig. 4. Motivation Example

B. Advanced Traveler

We extend our Basic *Traveler* algorithm to its advanced version in Algorithm 2. Algorithm 2 only shows the difference between Advanced Traveler algorithm and its basic version in Algorithm 1, that are Line 4 and Line 10. Basically speaking, the only difference between Advanced *Traveler* and Basic *Traveler* is that: we only count real records (not pseudo records) in the condition (Line 4). When we determine the size of CL, we also only count real records (Line 10).

Algorithm 2 Advanced Travel

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

C. N-Way Traveler

As we all know, there exists less dominant relationship in high-dimension datasets. Therefore, Basic Traveler algorithm may suffer from the curse of dimensionality. In order to handle high-dimension datasets, we propose N-Way Traveler algorithm. Given a 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 sub-functions[1], that are $F(x_1...x_m) = G(f_1(x_1...x_{d_1})f_2(x_{d_1+1}...x_{d_2})...f_n(x_{d_{n-1}+1}...x_m))$, where x_i is a dimension. N-Way *Travel* algorithm travels n DG_i in parallel until that all un-accessed records cannot be in top-k answers. Traveling on each DG_i ranks records on the value of sub-function f_i. In fact, we combine *TA* algorithm and Basic *Travel* algorithm into the N-Way *Travel*. The pseudo codes of N-Way *Travel* are given in Algorithm 3. Initially, for each DG_i, the records in the first layer are computed by sub-function 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 $\beta = G(f_1(R_1|_{I_1})...f_n(R_n|_{I_n}))$, where $R_i|_{I_i}$ is the projection of R_i on dimension set I_i (Line 5 and 12). It can be proved that β is the upper bound of the scores for all un-accessed records. The algorithm is iterated until $\delta \geq \beta$ (Line 7-13). During the iteration, for each DG_i, the operation is analogous to Basic Traveler (Lines 8-11).

V. DG 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 *n*th convex hull layer, all *m*th layers need to be re-computed, where m $\leq$ n. Computing convex hull is very expensive [5], [16]. In PREFER, in order to handle insertion and deletion, we need to re-select and re-materialize views. In AppRI, authors also point out that it is advisable to perform index maintenance in batches [1][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.

[1]We assume that query function F is decomposable.

[2]Two temporal solutions for online maintenance of AppRI are suggested in [1]. For example, we can mark some deleted records. However, using temporal solutions, the quality of AppRI index will degrade with frequent updates. Authors in [1] point out that AppRI needs to be rebuilt periodically

Algorithm 3 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:     all records at 1st layer are computed by query function $F$ on all dimensions. They are inserted the $CL$.
 4: Update $\delta$ to be k-th largest score in $CL$.
 5: Set $\beta = G(f_1(R_1|_{I_i})...f_n(R_n|_{I_n}))$, 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:         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 $CL_i$.
 9:         we move the heap head $R_i$ from $CL_i$ to another max-heap $RS_i$.
10:         All $R_i$'s children whose all parents are in $RS_i$ now are collected to form the set $C_i$.
11:         All records in $C_i$ are computed by sub-function $f_i$ on dimension set $I_i$. They are inserted into the heap $CL_i$.
12:     Set $\beta = G(f_1(R_1|_{I_1})...f_n(R_n|_{I_n}))$, where $R_i$ is the record at the head of heap $CL_i$.
13:     Update $\delta$ to be k-th largest score in $CL$.
14: Report top-k answers in $CL$.
```

Algorithm 4 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 $n$th layer and $R$.
16: Build the "parent-children" relationships between records in $R$ and $n+2$th layer.
17: report the updated $DG$.
```

A. Insertion

Algorithm 4 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 1st layer of DG, R will be inserted into the 1st layer (Lines 1-2). Otherwise, if R is dominated by some records P_i in the 1st layer, we can find the longest path L from P_i to some record in the *n*th layer using DFS search (Depth First Search) from P_i and each record in the path L dominates R (Lines 3-6). 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 the *n*th layer with R (Line 15), and R with records in the *(n+2)*th layer (Line 16).

Time Complexity In Lines 1-6, we locate the level in DG where R should be inserted. We can always find the right level by accessing all records indexed by DG. Therefore, the complexity in Lines 1-6 is $O(|D|)$, where $|D|$ is the cardinality of record set. For Line 9, there are at most $|D|$ records in the set S, which can be found in $O(|D|)$ time. It also means that there are $O(|D|)$ loops in Lines 10-14. The time complexity in Line 12 is $O(|D|)$, since there are at most $O(|D|)$ records in the current next layer. The time cost in Line 13-14 is $O(|D|)$, since each record has $O(|D|)$ parents at most. Considering the loop, the complexity in Line 10-14 is $O(|D|^2)$. The time cost in Line 15-16 is also $O(|D|)$. In conclusion, the complexity of insertion algorithm is $O(|D|^2)$ in the worst case.

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

B. Deletion

When we delete some record R in the *n*th layer, the simplest solution is that we can mark the record R as a *pseudo record*. In this way, Advantage *Traveller* algorithm can still find the correct top-k answers.

We discuss another deletion algorithm of DG, shown in Algorithm 5. When we delete a record R, we first find the position of R in DG (Line 1). Then delete it from the *n*th 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 *n*th 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 *n*th 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).

Time Complexity If a record R in DG is deleted from DG, in the worst case, all its descendant records of R are affected by chain reaction in Line 7. The number of all descendant records is $O(|D|)$. For each affected record C_i, we need to promote it into the previous level and build the "parent-children" relationships between records at its current previous layer and C_i (Lines 5-6). The time complexity in Lines 5-6 is $O(|D|)$ at most. In conclusion, the complexity of deletion algorithm is $O(|D|^2)$ in the worst case.

VI. 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 [2], CA [2], Onion [5], PREFER [6], and some state art of top-k algorithms, such as AppRI [1], RankCube [17]. Since most top-k algorithms can only support linear query function, to enable fair performance comparison, we only use linear function in comparison study.

Experiment Setting. We implement TA, CA and Onion [3] by ourself. We download the software PREFER from the website: *http://db.ucsd.edu/prefer/*. The AppRI software is provided by authors in [1]. Since RankCube [17] supports both selection condition and ranking condition, in order to enable performance comparison, we ignore the selection condition in RankCube [17]. After discussion with authors in [17], we implement RankCube as follows by ourself: first partition the dataset into blocks according to Section 3.1.2 in [17], and then answer top-k query according to the query algorithm in Section 3.2.2 in [17]. All experiments coded by ourself are implemented by standard C++ and conducted on a P4 1.7GHz machine of 1G RAM running Windows XP.

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 dataset is 1,000K. For uniform datasets, attribute values are "uniformly distributed" between 0 and 1. For Gaussian datasets, 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. 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 3 numerical attributes (with cardinalities 569, 1855 and 256) to create a three-dimensional data set: *count*, *srv-count*, *dest-host-count*.

Experiment 1. (*Compare Basic and Advanced Traveler*) We evaluate the optimization technique, that is pseudo record, in the experiment. We use the number of records that are accessed as the measure for *Travel* algorithm. In Advanced *Traveler* algorithm, we introduce pseudo records to reduce the cost. Due to space limit, only a subset of our experiments are present here. Observed from Fig. 5(a), 5(b) and 5(c), in 5 dimension data sets, Uniform (U_5), Gaussian (G_5) and Correlated (R_5), pseudo records lead to the great reduction in the cost.[4] When evaluating the cost, accessed pseudo records also count. Even though we count accessed pseudo records, Fig. 5 shows that the number of all accessed records in Advanced *Traveler* are much less than that in Basic *Traveler*, which confirms the efficiency of the pseudo record technique.

Experiment 2. (*Comparison with Existing Top-K algorithms*) In the experiment, we compare our methods with existing algorithms. Firstly, the performance of *Traveler* (that is Advanced *Traveler*) is compared with two representative layer-based top-k algorithms, such as *Onion* [5] and *AppRI* [1]. In the offline phase, observed from Fig. 6(a) and Fig. 6(b), we need the least construction time to build DG index. In the online phase, as shown in Fig. 6(c) and 6(d), the number of accessed records in *Traveler* algorithm is much smaller than that in *AppRI* and *Onion*. The query response time in Fig. 6(e) and Fig. 6(f) also confirm the performance of *Traveler*.

We also compare *Traveler* with some non-layer 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. 7, *Traveler* outperforms other algorithms in both the number of accessed records and query response time.

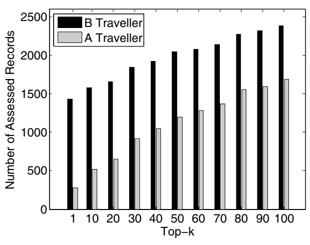
In experiments, we also compare *Traveler* with other methods in 5-dimension datasets. The results also confirm that our algorithm has a significant improvement. Due to space limit, we do not present here. For higher dimension (>5), we report our performance in Experiment 4. Furthermore, we observed that the increasing trend of the number of accessed records in *Traveler* is much slower than that in other methods in Fig. 6(c), 6(d), 7(a) and 7(b). Theorem 3.2 explains the cause. Given the query function F, the sets $\overline{S_2}$ 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.

Experiment 3 (*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 V and compare them with other layer-based algorithms. In Section V, we show that DG maintenance algorithms have

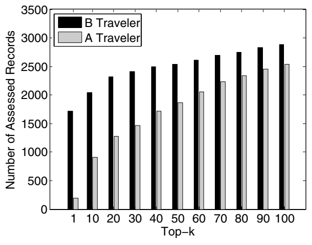
[3]In order to find convex hull layers in Onion algorithm, we use the software Qhull [16] downloaded from *http://www.qhull.org/*.

[4]Basic *Traveler* is denoted by "B *Traveler*" in Fig. 5. Advanced *Traveler* is denoted by "A *Traveler*" in Fig. 5, 6, 7 and 9

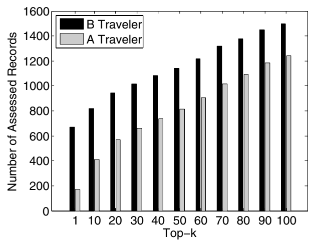
[5]RankCube is denoted by "RCube" in Fig. 7.



(a) U_5, $|D| = 1M$, $|m|$=5

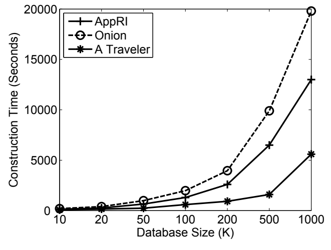


(b) G_5, $|D| = 1M$, $|m|$=5

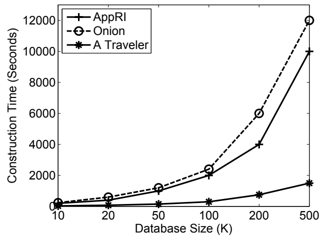


(c) R_5, $|D| = 1M$, $|m|$=5

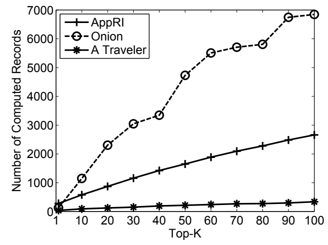
Fig. 5. *The Number of Accessed Records During Top-K Query*



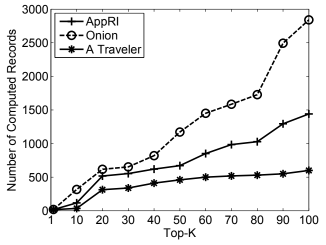
(a) Construction Time in U_3



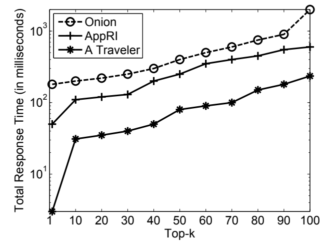
(b) Construction Time in Real dataset



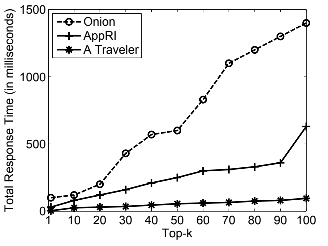
(c) Number of Accessed Records in U_3, $|D| = 1M$, $|m|$=3



(d) Number of Accessed Records in Real dataset, $|D| = 500K$, $|m|$=3

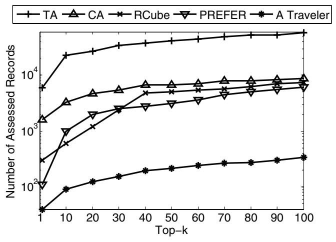


(e) Query Response Time in U_3, $|D| =$ $1M$, $|m|$=3

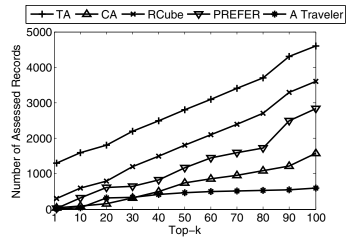


(f) Query Response Time in Real dataset, $|D| = 500K$, $|m|$=3

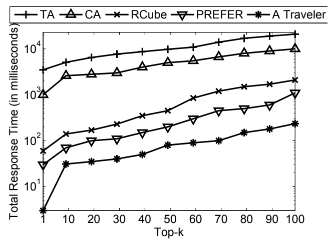
Fig. 6. *Comparison With Layer-Based Algorithms*



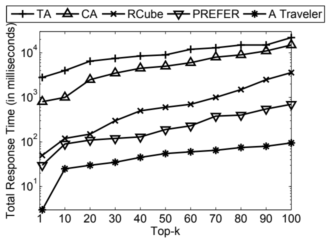
(a) Number of Accessed Records in U_3, $|D| = 1M$, $|m|$=3



(b) Number of Accessed Records in Real Dataset, $|D| = 500K$, $|m|$=3

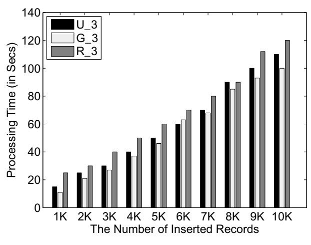


(c) Query Response Time in U_3, $|D| =$ $1M$, $|m|$=3

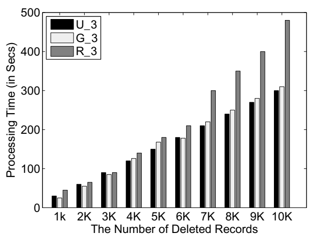


(d) Query Response Time in Real dataset, $|D| = 500K$, $|m|$=3

Fig. 7. *Comparison With Non-Layer Based Algorithms*



(a) *Insertion*



(b) *Deletion*

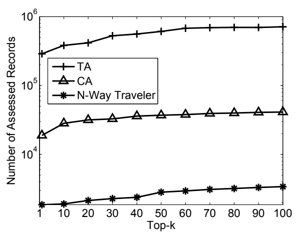
Fig. 8. *Evaluating DG Maintenance*

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. 8. Observed from Fig. 8, DG maintenance algorithm is suitable in the online process to handle insertion and deletion operations, since the running times are less than 140 seconds and 500 seconds for 10K insertions and deletions respectively.

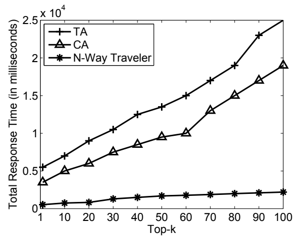
Fig. 8 does not show the maintenance cost in other layer-based algorithms, such as Onion and AppRI. As discussed in Section V, these algorithms need to re-construct layers to handle insertion and deletion. For example, in order to handle the same 10K insertions, we need about 19,000 and 13,000 seconds to re-construct layers in Onion and AppRI respectively in each 1,000K dataset. Therefore, our DG has the "light" maintenance cost.

Experiment 4 (*High Dimension and Worst Case Testing*) In the experiment, we first test N-Way *Traveler* algorithm in 10-dimension dataset. Most top-k algorithms are evaluated on the dataset with no more than 5 dimensions, such as Onion, PREFER, AppRI and RankCube. For example, in Onion, it is very difficult to find convex hull in a large dataset 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 5 dimensions respectively. Observed from Fig. 9(a) and Fig. 9(b), we find that N-Way *Traveler* outperforms TA and CA by orders of magnitude in both the number of accessed records and total response time.

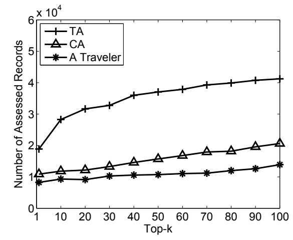
In our method, the worst case happens when all records are skyline points. In order to test Advanced *Traveler* algorithms in the special case, we generate a 5-dimension dataset, where all records are skyline points. Fig. 9(c) and 9(d) show that, due to pseudo record optimization technique, Advanced *Traveler* still works well in the worst case.



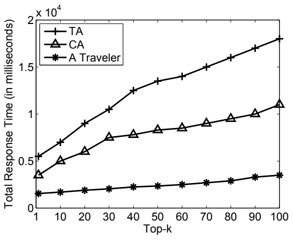
(a) Number of Accessed Records, $|m| = 10$, $|D|$= 1M



(b) Query Response Time, $|m| = 10$, $|D|$= 1M



(c) Number of Accessed Records in the worst case, $|D|$=100K, $|m| = 5$



(d) Query Response Time in the worst case, $|D|$=100K, $|m| = 5$

Fig. 9. *Evaluating Traveler in High-Dimension and the Worst Case*

VII. Related Work

Top-k preference queries (top-k queries for short) have been studied for a long time and many algorithms have been proposed. The first category is based on *ranked list* [3], [4], [2]. 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 [18], 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 [2] and LARA proposed by Mamoulis. et al. in [19]. 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 [5] and AppRI [1]. 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 (a) for any possible linear queries, t is not in top l-1 results; and (b) 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 *n*th layer, all records before the *n*th (including the *n*th 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 1st 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.

In relation database context, top-k query are also well studied [20][21][22]. In [20], Bruno et al. proposed a mapping strategy for top-k selection in RDBMS, i.e. a top-k query is translated into a single range query over RDBMS, which can be processed efficiently. In [21], Ilyas et al. proposed a rank-join algorithm in the relation database. Li et al. proposed RankSQL [22] that integrates ranking in database systems on both the logical algebra level and the physical implementation level. Furthermore, Jin et al. proposed KNN-based and Grid-based sweeping approach for top-k query [23]. In [17], Xin et al. proposed a new computational model, called *Rank Cube*. Different from the top-k query problem, both *selection* and *ranking condition* are considered together. In [24], Yi et al. proposed a top-k view maintenance method, which does not focus on top-k preference query. Furthermore, Soliman et al. discussed top-k query in uncertain database [25]

Parallel to top-k queries, there is another type of queries, *skylines*, which has attracted much attention recently. Given a data set *D* of dimension *d*, 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 *d* attributes of *X*, there exists at least one attribute value is smaller than the corresponding attribute in *Y* and the rest attributes of *X* are either smaller or equal to the matched attributes in *Y*. Please note that, the dominant concept in skyline query [8] is different from its counterpart in our paper. We use relationship $>$ (and $\geq$) instead of $<$ (and $\leq$). However, they are essentially equivalent to each other.

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 data set.

The skyline operator was first introduced by Borzonyi et al.in [8]. They proposed block nested loops (BNL) and divide-and-conquer (D&C) algorithm. In BNL, a list of skyline candidates is always maintained in the memory by scanning the whole data set. In D&C, we first divide the whole data set into several partitions. The partial skylines in each partitions are found by main memory algorithm. As last, the final skylines are found by "merging" by these partial skylines.

In [10], Tan et al.proposed two algorithms: Bitmap and Index. In bitmaps, all information required to decide whether a point is in the skyline is encoded by the bitmap. In "index" method, we use d lists to organize a set of d-dimensional points. In each list, the points are sorted in ascending order of their minimum coordinate. Benefitting by bitwise operation and the sorted lists, the two methods can find the skylines quickly.

In [11], Kossmann et al. proposed Nearest Neighbor (NN) method to process skyline queries progressively. In NN, we first find the point R with minimum distance from the coordinate origin. Based on R, we can divide the space into different partitions. Several partitions can be safely pruned. The remaining partitions are inserted into a *to-do* list. We repeat the above process until to-do list is empty.

Furthermore, a new progressive algorithm named Branch-and-Bound Skyline (BBS) is proposed by Papadias et al. in [9]. BBS starts from the root of R-tree, and insert all its children in a heap sorted according to the distance from the coordinate origin. Then the first entry in the heap is chosen to "expand", that is, all its children are inserted into the heap. Repeat the above "expand" approach until we reach some leaf node P of R-tree, which is a skyline. Some entries in the heap dominated by P are pruned out safely from the heap. Iteratively, we can find all skylines until heap is empty.

In fact, to build our DG, iteratively, we can use any above skyline algorithm to find all layers. Then we build the "parent-chilren relationship" between *i*th and *(i+1)*th layer.

VIII. 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 *pseudo records* 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.

Acknowledgements The work was done when the first author visited Hong Kong University of Science and Technology as a visiting scholar. The work was supported by Hong Kong RGC Grant No. 611907, National Grand Fundamental Research 973 Program of China under Grant No. 2006CB303000. The authors thank Dr. Dong Xin and Prof. Jiawei Han for providing software AppRI. The authors also thank the anonymous reviewers for their valuable comments.

References

[1] D. Xin, C. Chen, and J. Han, "Towards robust indexing for ranked queries." in *VLDB*, 2006.
[2] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS*, 2001.
[3] S. Nepal and M. V. Ramakrishna, "Query processing issues in image (multimedia) databases." in *ICDE*, 1999.
[4] U. Güntzer, W.-T. Balke, and W. Kießling, "Optimizing multi-feature queries for image databases." in *VLDB*, 2000.
[5] 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." in *SIGMOD*, 2000.
[6] V. Hristidis, N. Koudas, and Y. Papakonstantinou, "Prefer: A system for the efficient execution of multi-parametric ranked queries." in *SIGMOD*, 2001.
[7] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis, "Answering top-k queries using views." in *VLDB*, 2006.
[8] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator." in *ICDE*, 2001.
[9] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An optimal and progressive algorithm for skyline queries." in *SIGMOD*, 2003.
[10] K.-L. Tan, P.-K. Eng, and B. C. Ooi, "Efficient progressive skyline computation." 2001.
[11] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: An online algorithm for skyline queries." 2002.
[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. The MIT Press, 2001.
[13] S. Chaudhuri, N. N. Dalvi, and R. Kaushik, "Robust cardinality and cost estimation for skyline operator." in *ICDE*, 2006.
[14] P. Godfrey, "Skyline cardinality for relational processing." in *FoIKS*, 2004.
[15] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.
[16] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Trans. on Mathematical Software*, 1996.
[17] D. Xin, J. Han, H. Cheng, and X. Li, "Answering top-k queries with multi-dimensional selections: The ranking cube approach." in *VLDB*, 2006.
[18] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum, "IO-Top-k: Index-access optimized top-k query processing." in *VLDB*, 2006.
[19] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung, "Efficient aggregation of ranked inputs." in *ICDE*, 2006.
[20] N. Bruno, S. Chaudhuri, and L. Gravano, "Top-k selection queries over relational databases: Mapping strategies and performance evaluation," *ACM TODS*, 2002.
[21] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Supporting top-k join queries in relational databases." in *VLDB*, 2003.
[22] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song, "Ranksql: Query algebra and optimization for relational top-k queries." in *SIGMOD*, 2005.
[23] W. Jin, M. Ester, and J. Han, "Efficient processing of ranked queries with sweeping selection." in *PKDD*, 2005.
[24] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen, "Efficient maintenance of materialized top-k views." in *ICDE*, 2003.
[25] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang, "Top-k query processing in uncertain databases," in *ICDE*, 2007.