Efficient Approximate Top-*k* Query Algorithm Using Cube Index

Dongqu Chen, Guang-Zhong Sun[1], Neil Zhenqiang Gong[2]
Key Laboratory on High Performance Computing, Anhui Province
School of Computer Science and Technology
University of Science and Technology of China
cdq2012@mail.ustc.edu.cn, gzsun@ustc.edu.cn, neilz.gong@berkeley.edu

Abstract. Exact top-*k* query processing has caught much attention recently because of its wide use in many research areas. Since missing the truly best answers is inherent and unavoidable due to the user's subjective judgment, and the cost of processing exact top-*k* queries is highly expensive for datasets with huge volume, it is intriguing to answer approximate top-*k* query instead. In this paper, we define a novel kind of approximate top-*k* query, called *μ-approximation* top-*k* query, and introduce an efficient indexing structure, cube index, to support this query. Based on cube index, we propose our novel algorithm: Cube Index Algorithm (CIA). We analyze the complexity of both setting up μ-cube index and CIA algorithm. Moreover, extensive experiments show that the CIA has significant improvement on the performance, compared with the well-known approximate top-*k* query algorithm, TA_θ algorithm.

Keywords: Database query processing, Algorithms, Indexes.

1 Introduction

Exact top-*k* query processing has gained more and more attention recently because of its wide use in many fields, such as information retrieval, multimedia databases, P2P and sensor networks, etc. The main reason for such attention is that top-*k* queries avoid overwhelming the user with large numbers of uninteresting answers which are resource-consuming.

However, two main reasons convince us to abandon exact top-*k* query processing. First, the top-*k* query concept is heuristic anyway. Hardly any user is interested in all the exact *k* answers of a top-*k* query. Instead, they may be only interested in one or several relevant objects in the top-*k* (e.g. 500 or 2000) answers. So, due to the subjective judgment of the user, missing the truly best answers is inherent and unavoidable. This argument enlightens us to relax exact top-*k* query to approximate top-*k* query. Second, the cost of processing exact top-*k* queries is highly expensive for datasets with huge volume, and the size of datasets in practice is always quite huge. So it's intriguing to answer approximate top-*k* query instead of exact top-*k* query.

To solve the approximate top-*k* queries, Fagin propose the TA_θ algorithm in [3] based on the TA algorithm. Some papers have tried to reduce the cost of the query while improving the precision of the answers. Based on TA_θ, Theobald et al. [6] introduced a scheme to associate probabilistic guarantees with approximate top-*k*

[1]Corresponding author
[2]Neil Z. Gong is now a postgraduate in EECS Department, UC Berkeley. This work was completed when Neil Z. Gong was undergraduate student of USTC.

answers. In [8], Amato used a *proximity* measure to decide if a data region should be inspected or not. Only data regions whose *proximity* to the query region is greater than a specified threshold are accessed. This method is used to rank the nearest neighbors to some target data object in an approximate manner. Approximate top-k query processing has been also studied in peer-to-peer environments. The KLEE system (Michel et al. [2]) addressed this problem, where distributed aggregation queries are processed based on index lists located at isolated sites. KLEE assumes no random accesses are made to index lists located at each peer. Message transfers among peers are reduced by encoding messages into lightweight Bloom filters representing data summaries.

In this paper, we define a novel approximation to the top-k answers, the *μ-approximation*, and introduce an efficient indexing structure called cube index to support such *μ-approximation* top-k query processing. Based on cube index, we propose a novel algorithm: Cube Index Algorithm (i.e. CIA), an approximate top-k query algorithm using cube index on the database, to solve this problem.

The rest of this paper is organized as follows: First, we define the computation model formally and review the TA_θ algorithm in Section 2. In Section 3, we describe our method on setting up the cube index and then analyze its time complexity. Based on these, we show our algorithm CIA and analyze its cost in Section 4. Thereafter, we show the experimental results in Section 5. Finally, in Section 6, we conclude this paper and introduce our future work.

2 Computation Model and TA_θ Algorithm

In this section, we describe the model of top-k problem and review the TA_θ algorithm [3].

Our model of the dataset can be described as follows [3]: assume the database D consists of n objects, which are denoted as $x_1, x_2 \dots x_n$. Each object x is an m-dimensional vector $(s_1(x), s_2(x) \dots s_m(x))$, where $s_i(x)$ is the *ith* local score of x as a real number in the interval [0, 1]. For a given object x, x has a total score of $f(x)= f(s_1(x), s_2(x) \dots s_m(x))$, where the m-dimensional aggregate function f is supposed to be increasingly monotonic:

Threshold Algorithm with *θ-Approximation* (TA_θ)

Pre-computing Phase:

For each attribute $i \in \{1, 2 \dots m\}$, get every $s_i(x_j)$ where $j \in \{1, 2 \dots n\}$ and insert them into a sorted list L_i. Sorted list means that objects in each list are sorted in descending order by the $s_i(x_j)$ value.

Computing Phase:

1: Do sorted access in parallel to each of the m lists. As an object is seen through sorted access in some list, do random access to the other lists to find all its remaining local scores, and compute its overall score. Maintain a set Y containing the k objects whose overall scores are the highest among all the objects seen so far.

2: For each list L_i, let $\underline{s_i}$ be the last local score seen under sorted access in L_i. Define the *threshold value* τ to be $\tau = f(\underline{s_1}, \underline{s_2} \dots \underline{s_m})$..

3: Halt when $\theta \cdot M_k \geq \tau$, where $M_k = \min\{ f(x) \mid x \in Y\}$.

Fig. 1. Threshold Algorithm with *θ-Approximation*

Definition 2.1 *Aggregate Monotone Function* [3]. An aggregate function f is *monotone* if $f(a_1, a_2 \dots a_m) \le f(a_1', a_2' \dots a_m')$, whenever $a_i \le a_i'$ for every i.

In this paper, we assume the aggregate function is weighted summation function, $f(x) = \sum_{i=1}^{m} w_i s_i(x)$, where $s_i(x) \in [0, 1]$ and $\sum_{i=1}^{m} w_i = 1$ ($w_i \ne 0$). Our task is to determine the top-k objects, that is, k objects with the highest total scores. For approximate top-k query, Fagin et al. [3] defined a *θ-approximation* to the top-k answers:

Definition 2.2 *θ-Approximation* [3]. Let Y be a collection of k objects such that for each y among Y and each z not among Y, there are $\theta f(y) \ge f(z)$, where $\theta > 1$. Then Y is one of the top-k answers with *θ-approximation* and θ is the *relative approximation coefficient*.

To solve the *θ-approximation* top-k query, Fagin et al. [3] proposed the TA_θ algorithm, based on the threshold algorithm (i.e. TA). TA_θ is described in Fig. 1.

3 Cube Index

Before proposing our algorithm, we first introduce an efficient indexing structure called cube index to support such μ-approximation top-k query processing.

3.1 Description of Cube Index

We map the database to an m-dimensional hyperspace $[0, 1]^m$; each object x_j with scores $(s_1(x_j), s_2(x_j) \dots s_m(x_j))$ in the database is mapped to an m-dimensional point $p_j = (s_1(x_j), s_2(x_j) \dots s_m(x_j))$ in $[0, 1]^m$. We will not distinguish between object x and its corresponding point p discussed below. Similarly, $s_i(p)$ is the value of p's ith dimension and $f(p)$ is p's total score.

Now we define a *μ-approximation* to the top-k answers.

Definition 3.1 *μ-Approximation*. Let Y be a collection of k objects such that for each y among Y and each z not among Y, there are $f(y)+\mu \ge f(z)$, where $0 < \mu \le 1$. Then Y is one of the top-k answers with *μ-approximation* and μ is the *proportional approximation coefficient*.

Definition 3.2 *Dominate* [7]. Point p_1 *dominates* point p_2 if and only if for each $i \in \{1, 2 \dots m\}$, $s_i(p_1) \ge s_i(p_2)$ and there exists at least one member j of $\{1, 2 \dots m\}$ satisfying $s_j(p_1) > s_j(p_2)$.

Observation 3.1. If point p_1 *dominates* point p_2, then $f(p_1) > f(p_2)$, where f is an *aggregate monotone function*.

Proof. We can easily get the correctness of Observation 3.1 according to the definitions of *aggregate monotone function* and *dominate*. □

Definition 3.3 *Skyline* [7]. The *skyline* of a dataset D is the set of points that are not *dominated* by any point in D.

Definition 3.4 *Bottom Point*. The *bottom point* of a hypercube is the vertex whose values of every dimension are all lowest in the hypercube.

For example, the *bottom point* of the 3-dimensional cube $[0.2, 0.3] \times [0.1, 0.2] \times [0.5, 0.6]$ is $(0.2, 0.1, 0.5)$.

Observation 3.2. All other points in a hypercube *dominate* the *bottom point*.

Proof. We can easily get the correctness of Observation 3.2 according to the definitions of *dominate* and *bottom point*. □

Now we show the cube partition method on the m-dimensional hyperspace $[0, 1]^m$, which is described as follows:

Firstly, we set the length of the edge of each hypercube as μ, where $\mu \in [0, 1]$. Then we divide the interval [0, 1] into several μ-segments from 1 to 0 until the rest is shorter than μ. Each dimension is divided in this way so that the m-dimensional hyperspace $[0, 1]^m$ is partitioned into several hypercubes or sub-hyperspaces. Thereafter, we classify all the points in database into several sets: Point p_i belongs to bp_i's associated point set S_i if and only if p_i is in the hypercube whose *bottom point* is bp_i.

We call this partition method the *μ-cube partition*.

Definition 3.5 *Sky Point*. For a *μ-cube partition*, the *sky point* is the point whose values in every dimension are all $1 - \mu$, that is, the point $(1 - \mu, 1 - \mu\ldots 1 - \mu)$.

Apparently, *sky point* is the very *bottom point* which *dominates* all the other *bottom points* and the set {*sky point*} is the *skyline* of the set of *bottom points*.

Definition 3.6 *Neighbor*. *Bottom point* bp_1 is a *neighbor* of *bottom point* bp_2 if and only if they satisfy $\sum_{i=1}^{m}\left\lceil \frac{|s_i(bp_1)-s_i(bp_2)|}{\mu} \right\rceil = 1$.

Definition 3.7 *Superior*. *Bottom point* bp_1 is a *superior* of *bottom point* bp_2 if and only if bp_1 is a *neighbor* of bp_2 and bp_1 *dominates* bp_2.

Definition 3.8 *Inferior*. *Bottom point* bp_1 is an *inferior* of *bottom point* bp_2 if and only if bp_1 is a *neighbor* of bp_2 and bp_1 is *dominated* by bp_2.

Discussions on special cases:

1) For the points in the hypercube whose *bottom point* is the *sky point* belong to the 0^{th} set S_0.

2) The points on the intersecting hyperplane of several neighboring hypercubes belong to the hypercube whose *bottom point dominates* the others' *bottom point*.

3) The points coinciding with bp_i belong to set S_i.

4) If $S_i.size = 0$ and $i \neq 0$, then remove bp_i from the set of *bottom points*. Meanwhile, for each *inferior inf* of bp_i, regard all the *superiors* of bp_i as *inf*'s *superiors* too; for each *superior sup* of bp_i, regard all the *inferiors* of bp_i as *sup*'s *inferiors* too.

Definition 3.9 *μ-Cube Index*. For a *μ-cube partition*, the *μ-cube index* is an index list or array whose entries are ids of the *bottom points*. Each *bottom point* bp_i has its associated point set S_i as well as its *superiors*' ids and *inferiors*' ids.

3.2 Complexity Analysis of μ-Cube Indexing Method

Now we analyze the time complexity of the method on setting up the cube index, which is done in the pre-computing phase.

According to the description, the most time-consuming calculations in a *μ-cube partition* are to find the *superiors* and *inferiors* of each *bottom point* and to classify all the points in database into their corresponding sets.

Actually, the *superiors* and *inferiors* of each *bottom point bp* can be determined by the following two simple formulas:

1. For each $i \in \{1, 2 \dots m\}$ and $s_i(bp) \neq 0$, *bottom point bp'* is one *inferior* of *bp*, satisfying
   a. $s_i(bp') = (s_i(bp)-\mu)\cdot H(s_i(bp)-\mu)$, where $H(x)$ is the Heaviside step function;
   b. $s_j(bp') = s_j(bp)$ for each $j \in \{1, 2 \dots m\}$ and $j \neq i$.
2. *Bottom point bp'* is one *inferior* of *bp* if and only if *bp* is one *superior* of *bp'*.

There are $\left\lceil \frac{1}{\mu} \right\rceil^m$ *bottom points* in total, so the time complexity to find the *superiors* and *inferiors* of each *bottom point* is $O\left(\left\lceil \frac{1}{\mu} \right\rceil^m \times m\right)$.

On the other hand, each point *p* in database belongs to set S_i if and only if set S_i's corresponding *bottom point* bp_i satisfies that for each $i \in \{1, 2 \dots m\}$,

a. $s_i(bp) = (1-\left\lceil \frac{1-s_i(p)}{\mu} \right\rceil \times \mu) \times H(1-\left\lceil \frac{1-s_i(p)}{\mu} \right\rceil \times \mu)$ if $s_i(p) \neq 1$, where $H(x)$ is the Heaviside step function;

b. $s_i(bp) = 1 - \mu$ when $s_i(p) = 1$.

Similarly, there are *n* points in database, so the time complexity to classify all the points in database into their corresponding sets is O(*mn*).

Therefore, the total time complexity in the pre-computing phase is $O\left(m\left\lceil \frac{1}{\mu} \right\rceil^m + mn\right)$.

4 The Cube Index Algorithm

4.1 Description of Cube Index Algorithm

Based on the *μ-cube index*, we now propose a novel algorithm to answer the *μ-approximation* top-*k* query: the Cube Index Algorithm (i.e. CIA), which is described by the pseudo-code in Fig. 2.

Here *Selectively Add* in the pseudo-code is a sub- method to improve the precision of the algorithm qualitatively. It can be to add the points at random, or to add them from the points in *skyline* of S_i or others ways.

4.2 μ-Approximation of Cube Index Algorithm

To proof the *μ-approximation* of CIA, we first introduce three lemmas and a corollary as follows.

Lemma 4.1. Set *T* is always the top-(*T.size*) answers to the set of *bottom points*.

Proof. (By mathematical induction) *Basis:* Set *T* = {*sky point*} is the top-1 answers to the set of *bottom points*. Actually, *sky point dominates* all the other *bottom points* for the formula of *μ-cube index* and the definition of *sky point*. According to Observation 3.1, the *sky point* is the top-1 in the set of *bottom points*.

Cube Index Algorithm (CIA)

Pre-computing Phase:

Execute the normalization then set up the *μ-cube index* on the database.

Computing Phase:

```
Y = ∅, CL = ∅, T = {sky point}, where Y is the result set while CL is the sorted
   candidate list according to the total scores and T is a temp set.
if S_0.size ≤ k then
    add all points in S_0 into Y
else
    Selectively Add k points in S_0 into Y.
bp_i = sky point.
while (Y.size < k) do
    for each inferior inf of bp_i do
        if inf has not been accessed before and all superiors of inf is among T then
            Access inf and insert it into CL
        else
            Continue.
    if CL.size > k − Y.size then
        Only keep the first k − Y.size points in CL.
    Let bp_i be the bottom point with the highest score in CL and move it into T.
    if S_i.size ≤ k − Y.size then
        add all points in S_i into Y
    else
        Selectively Add k − Y.size points in S_i into Y.
Return Y.
```

Fig. 2. Cube Index Algorithm

Inductive step: Assume that set T is the top-j answers to the set of *bottom points* now, then the *bottom point* bp_i with the highest score in CL is the top-(j + 1) in the set of *bottom points* and is supposed to be moved to set T from CL.

Actually, only the points in the CL now have the chance to be the top-(j + 1). Otherwise, for a point bp which is not in CL or set T, either bp has been accessed before or bp has at least one *superior* that is not in set T. In the first case, according to the algorithm, CIA halts if and only if $Y.size = k$, so $Y.size < k$ before the algorithm halts. If bp has been accessed before and be removed from CL, then there exist at least $T.size+(k-Y.size) \geq T.size+1 = j + 1$ points whose total scores are higher than bp so that bp even has no chance to be one of the top-(j + 1) answers. In the other case, according to the definition of *superior* and Observation 3.1, every *superior* sup of bp satisfies $f(sup) > f(bp)$, so once sup is not in the top-j answers, or set T, bp has no chance to be one of the top-(j + 1) answers. Furthermore, for each point bp in CL, where $bp \neq bp_i$, bp is impossible to be one of the top-(j + 1) answers because even bp_i is not in the top-j answers. Therefore, bp_i is the top-(j + 1) in the set of *bottom points*.

Conclusion: When CIA halts, set T is the top-($T.size$) answers to the set of *bottom points*. □

Corollary 4.1. *Bottom points* are moved into set T in descending order of total score.

Proof. From the proof of Lemma 4.1, we easily conclude that *bottom points* are moved into set T in descending order of total score. □

Lemma 4.2. When CIA halts, there is at most one *bottom point* bp_j in set T satisfying $S_j \not\subseteq Y$, where bp_j is the one with the lowest score in set T and for each $bp_i \in T$ and $bp_i \neq bp_j$, $S_i \subseteq Y$.

Proof. According to the algorithm, the sub-method *Selectively Add* is executed if and only if $S_j.size > k - Y.size$. In this case, we *Selectively Add* $k - Y.size$ points in S_j into Y so that $S_j \not\subseteq Y$. Thus there would be $Y.size = k$ once the *Selectively Add* has been executed, where CIA halts. So the sub-method *Selectively Add* can be executed at most once. For Corollary 4.1, bp_j is the one with the lowest score in set T. However, in the case that $S_i.size \leq k - Y.size$, we add the whole S_i into set Y so that $S_i \subseteq Y$.

Therefore, when CIA halts, there is at most one *bottom point* bp_j in set T satisfying $S_j \not\subseteq Y$, where bp_j is the one with the lowest score in set T and for each $bp_i \in T$ and $bp_i \neq bp_j$, $S_i \subseteq Y$. □

Lemma 4.3. For point $p_i \in S_i$ and point $p_j \in S_j$, if $f(bp_i) \geq f(bp_j)$, then $f(p_i)+\mu \geq f(p_j)$.

Proof. According to the formula of *μ-cube index* and the definition of *bottom point*, for each $l \in \{1, 2 \ldots m\}$, there is $s_l(bp_j) \leq s_l(p_j) \leq s_l(bp_j) + \mu$. Considering $f(x)=\sum_{l=1}^{m} w_l s_l(x)$, where $s_l(x) \in [0, 1]$ and $\sum_{l=1}^{m} w_l = 1$, we have

$$f(bp_j) \leq f(p_j) \leq \sum_{l=1}^{m} w_l \left[s_l(p_j) + \mu \right] = \sum_{l=1}^{m} w_l s_l(p_j) + \sum_{l=1}^{m} w_l \mu = f(bp_j) + \mu$$

for Observation 3.1 and Observation 3.2. We can also get $f(bp_i) \leq f(p_i)$ in the same way.

Therefore, $f(p_i)+\mu \geq f(bp_i) + \mu \geq f(bp_j) + \mu \geq f(p_j)$. □

Theorem 4.1. CIA based on *μ-cube index* finds the top-k answers with *μ-approximation*.

Proof. According to the algorithm, if $bp_i \notin T$, any member of S_i has no chance to be added into set Y. That is, for each $y \in Y$ and $y \in S_y$, there must be $bp_y \in T$. And from Lemma 4.1, we know that set T is the top-($T.size$) answers to the set of *bottom points*. For each point $z \notin Y$ and $z \in S_z$ and for each $y \in Y$ and $y \in S_y$, if $bp_z \notin T$, then $f(bp_y) \geq f(bp_z)$, so $f(y)+\mu \geq f(z)$ for Lemma 4.3. In the other case, if $bp_z \in T$, since $z \notin Y$, meaning $S_z \not\subseteq Y$, bp_z is the one with the lowest score in set T according to Lemma 4.2. So we also have $f(bp_y) \geq f(bp_z)$ and $f(y)+\mu \geq f(z)$.

Therefore, for each y among Y and each z not among Y, there is $f(y)+\mu \geq f(z)$. That is, CIA based on *μ-cube index* finds the top-k answers with *μ-approximation*. □

4.3 Cost Analysis of Cube Index Algorithm

According to Fagin et al. [3], the cost of the top-k query is proportional to the times of accessing or aggregating the objects. For the CIA, the cost is the number of *bottom points* accessed in the query.

First, let $\underline{bp}$ be the last *bottom point* added into set T. Denote $B_1 = \{sky\ point\} + \{bp \mid bp$ is a *bottom point* which is accessed in the query$\}$ and $B_2 = T - \{\underline{bp}\}$. According to Lemma 4.1 and Corollary 4.1, B_2 is the top-$(T.size - 1)$ answers to the set of *bottom points*.

Theorem 4.2. The cost of the CIA is *T.size* – 2 + *skyline*($\overline{B_2}$)*.size*, where $\overline{B_2}$ is the complementary set of B_2.

Proof. We only need to show that $B_1 = B_2$ + *skyline*($\overline{B_2}$). Actually, it can be proved by apagoge.

Case 1: If there exists $bp \in B_2$ + *skyline*($\overline{B_2}$) but $bp \notin B_1$, then we know *bp* is not the *sky point*.

sub-case 1: If $bp \in B_2$, since $B_2 = T - \{bp\} \Rightarrow B_2 \subset T$, according to the algorithm, *bp* has no chance to be added into set *T* if *bp* has not been accessed. So it will conflict with the algorithm.

sub-case 2: If $bp \in$ *skyline*($\overline{B_2}$), then all the *superiors* of *bp* is in B_2 because there is no any point in $\overline{B_2}$ *dominating bp* according to the definition of *skyline*. However, in CIA, all points whose all *superiors* are in *T* must be accessed before the CIA halts. As $B_2 \subset T$, *bp* must be accessed, which contradicts the assumption that $bp \notin B_1$.

Case 2: If there exists $bp \in B_1$ but $bp \notin B_2$ + *skyline*($\overline{B_2}$), then *bp* belongs to neither B_2 nor *skyline*($\overline{B_2}$). The fact that $bp \notin B_2$ indicates *bp* is not in the top-*(T.size* − 1*)* answers to the set of *bottom points* so *bp* is not the *sky point* because {*sky point*} is the top-1 answers. Therefore, *bp* has chance to be accessed if and only if all the *superiors* of *bp* are in B_2 for the algorithm. However, $bp \notin$ *skyline*($\overline{B_2}$), meaning that *bp* has at least one *superior* that is not in *T* so *bp* cannot be accessed. Thus the assumption has no chance to be true.

Therefore, $B_1 = B_2$ + *skyline*($\overline{B_2}$). Besides, since *skyline*($\overline{B_2}$) $\subseteq \overline{B_2}$, $B_2 \cap$ *skyline*($\overline{B_2}$) $= \emptyset$. So B_1*.size* = B_2*.size* + *skyline*($\overline{B_2}$)*.size*. Moreover, B_2*.size* = *T.size* – 1 and the cost of the CIA is B_1*.size* – 1, considering that the *sky point* is not accessed in the algorithm.

Therefore, the cost of the CIA is B_1*.size* – 1 = *T.size* – 1 + *skyline*($\overline{B_2}$)*.size* – 1 = *T.size* – 2 + *skyline*($\overline{B_2}$)*.size*. □

5 Experiments

In this section, we conduct extensive experiments to evaluate the performance of our algorithm. Our algorithm is implemented in C/C++ language. We perform our experiments on an 8-CPU server with 8GB shared memory and each CPU is 4-core Intel Xeon E5430 2.66GHz.

5.1 Turning μ-Approximation into θ- Approximation

According to the definitions of μ*- approximation* and θ*-approximation*, if set *Y* is the top-*k* answers with μ*-approximation*, for each *y* among *Y* and each *z* not among *Y*, there are $f(y)+\mu \geq f(z)$. So we have $(1+\frac{\mu}{f(y)})f(y) \geq f(z)$. Let $f(y)$ be the *kth* highest total score in set Y so that $\frac{\mu}{f(y)} \leq \frac{\mu}{\underline{f}(y)}$. Therefore, the *relative approximation coefficient* $\theta = \frac{\mu}{\underline{f}(y)}$, or $\mu = \underline{f}(y) \cdot \theta$.

In our experiments, we run the CIA over the databases to find the value of $\underline{f}(y)$ and then the TA_θ runs on *θ-approximation* of $\theta = \frac{\mu}{\underline{f}(y)}$. We choose the *μ-approximation* as the criterion of approximation to run our tests.

5.2 Evaluation Metrics

In our tests, the following measures are collected for efficiency comparison [6]:

accesses: the number of items accessed in the query without duplication;

precision: the fraction of top-k results in an approximate result that belongs to the exact top-k result;

recall: the fraction of top-k results in the exact result that were returned by the approximate top-k query;

rank distance: the *footrule distance* [14] between the ranks of the approximate top-k results and their true ranks in the exact top-k result, i.e., $\frac{1}{k}\sum_{i=1}^{k}\left|i - truerank(i)\right|$, where *truerank(i)* is the *i*th returned object's true rank in the exact top-k result.

score error: the absolute error between approximate and exact top-k scores, i.e.,

$$\frac{1}{k}\sum_{i=1}^{k}\left|totalscore_i^{(approx)} - totalscore_i^{(exact)}\right|,$$

where $score_i^{(approx)}$ is the total score of the *i*th object in the approximate top-k result while $score_i^{(exact)}$ is the total score of the *i*th object in the exact top-k result.

Because the *precision* and the *recall* have the same denominator k, they have identical values in our setup. We regard the *recall* as a formal measure in our tests, instead of *precision*.

5.3 Description of Datasets

We do experiments on two synthetic datasets. All generated local scores belong to the interval [0, 1]. The two synthetic datasets are produced to model different input scenarios. They are UI and NI respectively. UI contains datasets in which objects' local scores are uniformly and independently generated for the different lists. NI contains datasets in which objects' local scores are normally and independently generated for the different lists. For synthetic datasets, our default settings for different parameters are shown in Table 1. As mentioned above, approximate top-k queries are usually applied in the cases that the values of n is fairly large, which could cause considerable cost and delays to return the exact query answers. Therefore, in our tests, the default number of data items in each list is 1,000,000, i.e. n=1,000,000. Typically, users are interested in a small number of top answers, thus we set k = 500 as the default value of k, which is a tiny value compared with n. We set m as 3 since

Table 1. Default values of experimental parameters.

Parameters	Default Values
The number of objects, i.e. n	1,000,000
The number of lists, i.e. m	3
The number of results returned, i.e. k	500
The precision of results returned, i.e. μ	0.05
Aggregate function	$0.2s_1+0.3s_2+0.5s_3$

most previous works evaluate their algorithms on datasets with 3 lists like [4]. Finally, we set 0.05 as the default value of μ.

We run our tests with default precision ($\mu = 0.05$) and high precision ($\mu = 0.005$) over each dataset respectively. Furthermore, we run the algorithms on the datasets with large value of k (2000) to observe the effect of k on the performance.

For real datasets, we choose El Nino dataset[1] and Forest Cover (FC) dataset[2]. El Nino dataset contains 93935 objects and FC dataset contains 581012 objects. El Nino contains oceanographic and surface meteorological readings taken from a series of buoys positioned throughout the equatorial Pacific. The data is expected to aid in the understanding and prediction of El Nino/Southern Oscillation (ENSO) cycles. FC contains 581012 forest land cells (i.e. objects), having four attributes (i.e. lists): horizontal distance to nearest surface water features, vertical distance to nearest surface water features, horizontal distance to nearest roadways, and horizontal distance to nearest wildfire ignition points. For both real datasets, we choose 3 lists and normalize the dataset with the formula: $\frac{s_i(t)-Min}{Max-Min}$, where $s_i(t)$ is t's *ith* local score.

5.4 Experimental Results

Fig. 3 illustrates the experimental results where all the parameters are set as default values. Apparently, CIA has significant reduction on the number of accesses over every dataset. Compared with the TA_θ, CIA reduces more than 99% accesses during the query process. Apart from this, CIA is also dominant on other evaluation metrics, namely *recall*, *rank distance* and *score error* over every dataset but FC, where CIA is a little inferior to TA_θ on these aspects.

The experimental results shown in Fig. 4 when $k = 2000$ on each dataset are similar to the results when all the parameters are set as default values. From the results, we can see that CIA also has great reduction on the number of accesses compared with the TA_θ. In terms of the other aspects, CIA performs much better than TA_θ over every dataset except FC.

Fig. 5 shows us the experimental results where the parameters are set as default values except that μ, the precision of results returned is 0.005. Obviously, CIA is more efficient than TA_θ considerably but is transcended in other measures. Therefore, CIA has lower accuracy compared with TA_θ but still keeps its efficiency in the queries with high precision.

Summary: From all the experimental results, we know that CIA improves significantly not only on the number of accesses, but also on other evaluation metrics in the queries with default precision. In addition, we can also learn the fact that CIA still keeps its efficiency and accuracy when the value of k is considerable large.

[1]From UCI KDD. http://kdd.ics.uci.edu/databases/el_nino/el_nino.html
[2]From UCI KDD. http://kdd.ics.uci.edu/databases/covertype/covertype.html

However, CIA is not dominant on all the evaluation metrics over some datasets, like FC in our tests. Finally, in the queries with high precision, our algorithm is considerably superior to TA_θ on the number of accesses but have little advantage on other respects.

Results for UI	*accesses*	*recall*	*rank distance*	*score error*
TA$_\theta$	10527	0.50200	281.78800	0.008390
CIA	7	0.75600	88.404000	0.002428
Results for NI	***accesses***	***recall***	***rank distance***	***score error***
TA$_\theta$	10703	0.52600	242.084000	0.007883
CIA	7	0.76800	88.180000	0.002601
Results for EI	***accesses***	***recall***	***rank distance***	***score error***
TA$_\theta$	1890	0.29200	702.208000	0.006722
CIA	2	0.66600	124.584000	0.001354
Results for FC	***accesses***	***recall***	***rank distance***	***score error***
TA$_\theta$	5031	0.99200	0.506000	0.000017
CIA	61	0.83800	35.214000	0.001281

Fig. 3. Performance of CIA vs. TA_θ when $k = 500$ and $\mu = 0.05$

Results for UI	*accesses*	*recall*	*rank distance*	*score error*
TA$_\theta$	28778	0.77900	232.320500	0.003214
CIA	24	0.83700	158.745000	0.001843
Results for NI	***accesses***	***recall***	***rank distance***	***score error***
TA$_\theta$	29375	0.80200	194.234000	0.002834
CIA	26	0.85100	136.511000	0.001665
Results for EI	***accesses***	***recall***	***rank distance***	***score error***
TA$_\theta$	4519	0.70300	463.897000	0.006876
CIA	4	0.94750	17.667000	0.000258
Results for FC	***accesses***	***recall***	***rank distance***	***score error***
TA$_\theta$	10084	0.94150	23.150500	0.000418
CIA	138	0.89650	75.043000	0.001175

Fig. 4. Performance of CIA vs. TA_θ when $k = 2000$ and $\mu = 0.05$

Results for UI	*accesses*	*recall*	*rank distance*	*score error*
TA$_\theta$	40683	0.99800	0.030000	0.000001
CIA	532	0.97400	1.112000	0.000031
Results for NI	***accesses***	***recall***	***rank distance***	***score error***
TA$_\theta$	40371	0.99999	0.000001	0.000001
CIA	539	0.97800	0.678000	0.000023
Results for EI	***accesses***	***recall***	***rank distance***	***score error***
TA$_\theta$	8941	0.99999	0.000001	0.000001
CIA	22	0.96200	2.200000	0.000023
Results forFC	***accesses***	***recall***	***rank distance***	***score error***
TA$_\theta$	10482	0.99999	0.000001	0.000001
CIA	552	0.97400	0.840000	0.000027

Fig. 5. Performance of CIA vs. TA_θ when $k = 500$ and $\mu = 0.005$

6. Conclusions and Future Work

In this paper, we analyzed the model of the top-*k* queries and gave some observations. To measure the approximation of the top-*k* answers, we defined a novel approximation, *μ-approximation* to the top-*k* answers. Then we introduce an efficient indexing structure called *μ-cube index* to support this kind of approximate query. Based on the *μ-cube index* on the dataset, we proposed our algorithm, the Cube Index Algorithm to answer the *μ-approximation* top-*k* queries. The main advantage of CIA is that we choose the *bottom point* of a hypercube to approximately represent the points in the hypercube and run the algorithm to find the top-*T.size* in the set of *bottom points* so that the number of accesses can be reduced significantly. Extensive experimental results on both generated and real-world datasets show that our algorithm owns higher accuracy with less cost, compared with TA_{θ}.

In the future work, we plan to turn our algorithm into exact algorithm based on the cube index ideas.

Acknowledgments. This work is supported by the National Science Foundation of China under the grant [No. 60873210].

References

1. I. Ilyas, G. Beskales, M. A. Soliman: A Survey of Top-k Query Processing Techniques in Relational Database Systems. ACM Computing Surveys, 2008.
2. S. Michel, P. Triantafillou, G. Weikum: KLEE: A frame work for distributed top-k query algorithms. VLDB, 2005.
3. R. Fagin, A. Lotem M. Naor: Optimal aggregation algorithms for middleware. PODS, 2001.
4. Neil Z. Gong, G. Z. Sun: Parallel Algorithms for Top-k Query Processing. ACM SIGMOD, 2010.
5. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: Introduction to Algorithms. MIT Press, 2001.
6. M. Theobald, G. Weikum, R. Schenkel: Top-k Query Evaluation with Probabilistic Guarantees. VLDB, 2004.
7. L.Zou, L.Chen: Dominant Graph An Efficient Indexing Structure to Answer Top-K Queries. ICDE, 2008.
8. G. Amato, F. Rabitti, P. Savino, P. Zezula: Region Proximity in Metric Spaces and Its Use For Approximate Similarity Search. ACM Trans. Inform. Syst, 2003.
9. D. Xin, J. Han, H. Cheng, and X. Li: Answering Top-k Queries with Multi-Dimensional Selections: The Ranking Cube Approach. VLDB 2006.
10. R. Fagin, R. Kumar, D. Sivakumar: Comparing Top K Lists. ACM-SIAM SODA, 2003.
11. D. Donjerkovic. R. Ramakrishnan: Probabilistic Optimization of Top N Queries. VLDB 1999.
12. J. Hellerstein, P. Haas, H. Wang: Online Aggregation. ACM SIGMOD, 1997.
13. I. Ilyas, W. Aref, A. Elmagarmid: Supporting Top-K Join Queries in Relational Databases. VLDB, 2004.
14. M. Kendall, J.D. Gibbons: Rank Correlation Methods. Oxford University Press, 1990.
15. C. Re, N. Dalvi, D. Suciu: Efficient Top-K Query Evaluation on Probabilistic Data. ICDE, 2007.