6-2010

Efficient Processing of Exact Top-k Queries over Disk-Resident Sorted Lists

Hwee Hwa PANG
Singapore Management University, hhpang@smu.edu.sg

Xuhua DING
Singapore Management University, xhding@smu.edu.sg

Baihua ZHENG
Singapore Management University, bhzheng@smu.edu.sg

Noname manuscript No.
(will be inserted by the editor)

Efficient Processing of Exact Top-k Queries over Sorted Lists

HweeHwa Pang · Xuhua Ding · Baihua Zheng

the date of receipt and acceptance should be inserted later

Abstract The top-k query is employed in a wide range of applications to generate a ranked list of data that have the highest aggregate scores over certain attributes. As the pool of attributes for selection by individual queries may be large, the data are indexed with per-attribute sorted lists, and a threshold algorithm is applied on the lists involved in each query. The threshold algorithm executes in two phases – find a cut-off threshold for the top-k result scores, then evaluate all the records that could score above the threshold.

In this paper, we focus on exact top-k queries that involve monotonic linear scoring functions over disk-resident sorted lists. We introduce a model for estimating the depths to which each sorted list needs to be processed in the two phases, so that (most of) the required records can be fetched efficiently through sequential or batched I/Os. We also devise a mechanism to quickly rank the data that qualify for the query answer and to eliminate those that do not, in order to reduce the computation demand of the query processor. Extensive experiments with four different datasets confirm that our schemes achieve substantial performance speed-up of between two times and two orders of magnitude over existing threshold algorithms, at the expense of a memory overhead of 4.8 bits per attribute value. Moreover, our scheme is robust to different data distributions and query characteristics.

Keywords Top-k query processing, threshold algorithm, Bloom filter.

School of Information Systems
Singapore Management University
E-mail: {hhpang, xhding, bhzheng}@smu.edu.sg

1 Introduction

A top-k query produces a ranked list of the k highest scoring records from a dataset, in which the score of a record is aggregated over the query attributes. Where the schema of the dataset contains many attributes, one obvious method to support top-k query processing is to organize the data in a multi-dimensional index structure, as in [29]. This method works well for data that involve a small number of attributes, but not for datasets that contain many attributes because no existing index structures are known to perform efficiently beyond 5 or 6 dimensions [22]. A more scalable approach is to create an sorted list for each attribute, in which the records are ordered by that attribute, and to apply the threshold algorithm [14] over the attributes specified in each query.

In this paper, we focus on the problem of efficiently processing *exact* top-k queries involving monotonic linear scoring functions over disk-resident sorted lists. Our objective is to minimize the query processing time, comprising CPU computation and disk I/O costs, subject to available memory resources. Examples of the problem include retrieval of multimedia information based on feature vectors extracted from the object contents (e.g. [48]), as well as text search engines that order documents according to their similarity to the term composition of each user query [49].

The classical solution to the above problem is the threshold algorithm [14]. In this algorithm, query processing involves two phases – finding a cut-off threshold for the top-k result scores, followed by processing the sorted lists to enough depth so as to evaluate all the records that could accumulate enough score to match the threshold, and thus qualify as candidates for the

top-k result. To optimize the two phases, we investigate the following concerns:

1. From the query, how do we determine $depth_{thres}$, the depth to which to scan each sorted list in order to determine the cut-off threshold? With $depth_{thres}$, the entries in the sorted lists that are needed for phase one can be fetched through sequential I/Os if each list occupies contiguously disk blocks, as exemplified by text search engines [49]. Even if the sorted lists are not stored contiguously on disk, fetching them together still allows the disk to service the I/O requests according to the elevator (or SCAN) algorithm [34], thus reducing seek costs.
2. Having derived the cut-off threshold, how do we determine $depth_{result}$, the depth to scan the sorted lists to in phase two? With $depth_{result}$, the required entries on the sorted lists can be fetched through sequential or batched I/Os, as for phase 1.
3. Among the records that are encountered in the two phases, how do we *safely* prune away the non-viable ones that will not accumulate enough scores from the sorted lists to qualify for the top-k result, without incurring random I/Os to fetch their values? This question is pertinent because there could be a huge number of non-viable records, and an effective pruning mechanism would reduce the memory footprint and computation demands of the threshold algorithm.

Our solutions to the above concerns are summarized below:

- We introduce a model for determining, for any top-k query involving m sorted lists, the expected depths $depth_{thres}$ and $depth_{result}$ to which to scan the lists in order to generate the query answer. These two estimates enable the system to retrieve all the requisite records from each query list in two sequential/batched I/O operations, first to $depth_{thres}$ and then to $depth_{result}$, instead of incurring random I/Os in rotating through the query lists.
- We extend our model to handle general workloads that are likely to occur in practice, including non-identical list distributions, queries that specify different weights for the lists, and non-uniform distributions for the attributes. Our extended model retains the ability to estimate $depth_{thres}$ and $depth_{result}$, thus enabling our solution to support realistic applications efficiently by exploiting sequential I/Os.
- We propose a flexible organization that allows Bloom filters [6] on the sorted lists to be dynamically grouped into buckets that are sized according to the query characteristics. With the buckets, the query processor is able to derive with high confidence a tight upper-bound score for each candidate record. The upper-bound scores are used to quickly prune away candidate records that would ultimately fail to accumulate sufficient scores to qualify as top-k results, as well as to order those records that do qualify.
- We design a query processing algorithm, called TBB for Threshold algorithm over Bucketized sorted lists with Bloom filter, that takes advantage of the $depth_{thres}$ and $depth_{result}$ estimates, as well as the candidate pruning mechanism, to process top-k queries efficiently. If the estimated input depths turn out to be shallower than required, TBB will continue to scan down the query lists, till the cut-off threshold is found in the case of $depth_{thres}$, or till the entire top-k result is determined in the case of $depth_{result}$. Therefore the estimated input depths help to optimize performance, *without undermining the correctness of the query results.*
- We report results of extensive experiments, using four sets of synthetic and real workloads that exhibit very different properties, which confirm the usefulness of our $depth_{thres}$ and $depth_{result}$ estimates as well as candidate pruning mechanism. Overall, our scheme achieves speed-up of between two times and two orders of magnitude over existing threshold algorithms, at an average memory overhead of only 4.8 bits per record attribute value.

The rest of the paper is organized as follows. Section 2 reviews related work, and differentiates our scheme from the existing solutions. In Section 3, we formulate the addressed problem, then develop the TBB scheme. An empirical evaluation of the scheme is reported in Section 4. Finally, Section 5 concludes the paper.

2 Related Work

We begin this section by reviewing the threshold algorithms by Fagin et al. Following that, we examine several studies that enhanced the threshold algorithms, and that applied them to different types of systems. Finally, we summarize the differentiations of our work here from the existing solutions in the literature.

2.1 Threshold Algorithms

The problem of top-k query over sorted lists was first formalized in the seminal work of Fagin et al [13,14]. Given m repositories, each holding a sorted list of objects in decreasing scores on a different attribute, the threshold algorithm (TA) produces the query results as follows. TA cycles through the sorted lists, popping an

object from the front of the list each time. For each encountered object, TA immediately retrieves all the attribute values of the object in order to derive its score. The algorithm terminates when the threshold, which aggregates the last encountered score from all the lists, falls below the k-th highest object score. TA can potentially incur very high processing overheads arising from all the random accesses for the object attributes.

For systems in which random access is expensive, [14] proposes an NRA variant. NRA executes in two phases. In phase one, NRA cycles through the sorted lists, popping a candidate object from the front of each list in turn. Instead of issuing a random access for the attribute values immediately, NRA accumulates the score of a candidate object gradually, as its component scores are gathered from the sorted lists. Phase one ends when the threshold falls below the k-th highest accumulated object score. In the second phase, NRA does not entertain new candidate objects. Rather, it pops more entries from the sorted lists only to establish tighter bounds for the existing candidate objects. The algorithm terminates once it determines that there is total ordering among the k highest scoring objects, and that the remaining candidates cannot accumulate enough scores to displace those top-scoring objects.

2.2 Extensions of Threshold Algorithms

A variation of the threshold algorithm is presented in [15]. Here, Güntzer et al showed that the number of random object accesses can be reduced through more termination condition tests, as well as through heuristics that generally hasten the fulfillment of the termination condition. The proposed heuristics include scanning the top-ranked objects in all the sorted list at the start, then focusing on those lists that exhibit steeper declining scores thereafter.

The threshold algorithm has been applied to several problem settings. [27] extended it to top-k queries on databases that are accessed over the Web. As the databases are autonomous, their attributes may be probed only through certain restricted interfaces, rather than being exposed for direct manipulation by the queries. Indeed, some of the databases may not even offer a sorted view. In the paper, Marian et al proposed an Upper algorithm that determines, after each probe, whether to issue a sorted-access probe on a database to get new objects, or to perform a random-access probe for the current most promising object. The scheme is able to exploit parallel accesses to the databases in order to minimize response time.

In [9], Chang and Hwang addressed the problem of processing top-k queries that involve predicate evaluations. Some of the predicates might offer sorted access of the objects at very low cost (through index structures), whereas others might involve user-defined functions, external predicates and joins. As the latter entails costly per-object probes, an MPro algorithm was given that ensures every probe performed is necessary for finding the top-k query answers. The authors extended this work in [19], to a general framework for finding the algorithm that is optimal for the given resource characteristics, e.g., different cost trade-offs between sorted access and random access.

Another study, reported in [5], shows how to schedule index access steps in processing top-k queries. The proposed query processing scheme interleaves sorted accesses and random lookups to the sorted lists, with the objective of minimizing the weighted sum of all the sorted accesses and random accesses. Each sorted list is organized as a series of blocks, with the block size being a one-off configuration parameter that is chosen to balance disk seek time and transfer rate. Furthermore, a histogram of the value distribution is maintained for each term, for estimating the probability that a given record can qualify for the top-k result, and hence whether to incur random accesses to probe its values. Experiments showed that the scheme outperformed the original threshold algorithms by a factor of 1.5 to 3 times in most cases, and up to 5 times at the maximum. This work does not estimate the required depths to take advantage of sequential or batched I/Os for fetching the query lists, which could lead to substantial incremental speed-ups as we will demonstrate. It also does not perform candidate pruning as done in our scheme, which entails a quick and definite, not probabilistic, determination of the viability of each candidate without looking up its values.

[2] proposed a couple of techniques to optimize the TA algorithm. The first technique tracks the deepest position in each sorted list, before which all the entries have been seen through either sorted or random accesses; these are the "best positions". By constituting the threshold from the scores at the best positions, rather than those at the last sorted access in each list, TA is likely to be able to terminate earlier. The second technique tracks the data objects that have been examined through sorted or random accesses, so as to avoid refetching them. These techniques substantially reduce the execution costs, and are incorporated in the TA implementation of our study.

In [25, 26], Mamoulis et al proposed an efficient realization of the NRA algorithm, called LARA. LARA is based on the observation that in phase one of the NRA algorithm, the set of candidate objects can only grow and it is no beneficial to attempt any pruning. Thus,

LARA provides data structures to efficient maintain (i) the objects seen so far, along with their partial aggregate scores and the input lists that have produced each object, (ii) the top-k objects with the highest lower-bound scores, and (iii) the latest score encountered on each input list. For phase two, the aim is to avoid updating the upper-bound score of every candidate object each time an object is read off the input lists. Instead, LARA maintains, for each combination of input lists, the identifier of the object that satisfies these conditions: (i) the object has been seen only on these lists, (ii) the object has the highest partial aggregate, and (iii) the object is not among the top-k scoring ones currently. Experiments confirm that LARA has much better time and space complexities than the original NRA. The techniques in LARA complement our solution nicely. Indeed, we implement our proposed schemes on top of LARA.

2.3 Approximate Top-k Queries

In [39], Theobald et al proposed a set of statistical techniques to estimate the score of candidate objects, with the aim of determining when it is safe to drop candidates that, with high probability, will not qualify for the query result.

[28] shows that, for approximate top-k queries, histograms on the value distribution in the various sorted lists can be used to estimate the aggregate score of candidate records. Moreover, a Bloom filter is built on the record identifiers in each cell of a histogram, for detecting quickly whether a given record falls within the cell. The authors did not address the issue of cell size configuration; as we will show, the best cell size varies from query to query, and we need a flexible scheme that can support different cell sizes according to the characteristics of individual queries.

Another study along this line of research is [33], which provides techniques to produce best-effort results subject to given resource constraints. The aim there is to maximize the quality of the query results.

A common characteristic of this category of work is that It is possible for a true top-k result to be pruned wrongly, so the techniques are not appropriate where exact top-k results are required.

2.4 Applications of Threshold Algorithms

In the context of relational databases, the top-k query corresponds to a ranked join on the input tables. Ilyas et al [20] proposed to implement the rank join algorithm as a pipelined operator, and presented formulas for estimating the depth to which the two operand tables need to be scanned for any desired result size k. [8] developed a cost model for choosing which of the available indices to use, in order to effect TA-like strategies to answer top-k queries in a relational DBMS.

In [32], Schnaitter et al studied the problem of estimating, for the query plan of a rank join operation, the input depths of its operand relations that are needed to produce the k top-scoring result tuples. Representing the joint frequency distribution of the inputs by tensors, the method repeatedly pulls the highest scoring tensor from the output, until the cumulative frequency of the pulled tensors reaches k. The score of the last tensor gives the cut-off threshold, which is used to calculate the depth of each input. In other words, this work estimates $depth_{result}$ but not $depth_{thres}$. It also does not deal with pruning the non-viable records within $depth_{result}$ of the input relations.

[11] studied the problem of top-k query processing over multimedia repositories. Here, the system is capable of performing a graded search for a feature/attribute selection condition that returns the objects matching the condition closely, as well as probing for the grade of match of a given object with respect to a condition. The authors described a scheme for determining which conditions should be evaluated through graded search, and which should be probed.

The threshold algorithm is a natural fit for text search engines. In these systems, the corpus is typically represented as an inverted index that maps a search term to a ranked list of documents that contain that term, and the answer for a search query comprises those documents that have the highest weighted sum over the search terms [49]. In [30], Ntoulas and Cho described a number of techniques for pruning terms and documents from the inverted index, in such a way that most of the queries can be satisfied from the pruned index without degrading the quality of the query results.

2.5 Other Related Work

Several variations of the top-k query problem have also been studied in the literature. One variation involves indexing schemes other than sorted lists. [43] proposed a solution for top-k aggregate query over indexed relations. [41] investigated top-k queries over datasets that are horizontally distributed in large-scale peer-to-peer networks. In [12], Deshpande et al introduced the AL-Tree, a multi-dimensional index, which is shown to offer efficient pruning of the search space during query processing. In [50], Zou and Chen proposed to build a Dominant Graph structure to facilitate the efficient processing of top-k queries. [37] introduced the SUBSKY

technique, which transforms the attributes of data objects into one-dimensional values; the transformed values can then be indexed with the B-tree to support subspace top-k queries.

[35] formulated a probabilistic formulation of top-k queries over data that may be imprecise and uncertain; here, the problem is to find the list of data objects that have the highest probability of being top-k. In [44], Yi et al considered top-k queries over uncertain databases in which each x-tuple may instantiate randomly into one tuple from multiple possibilities. [3] addressed the challenge of reporting, at anytime during the execution of the TA algorithm, the confidence that the top-k result has been found.

Other studies have extended top-k processing to typicality queries ([16,17]), dominating queries ([45,46], [24]), data streams ([18],[21]), twig queries on weighted graphs ([31]), semistructured data ([38]), spatial cum aggregation queries ([47]), and queries for pairs of similar records ([42]). Due to the difference in the problem setting, data structure, processing strategy and resource considerations, the solution approaches in those papers are different from our work here.

2.6 Differentiation of Our Work

To the best of our knowledge, no existing work has carried out a comprehensive optimization of the threshold algorithm that covers its total resource demands. The value of this work is that we provide a unified solution that includes techniques for optimizing the I/Os, memory and computation demands. In addition, our model for determining the required depths of the query lists works for both phases of the threshold algorithm, as well as for a varied range of data distributions, and hence is more general than the existing work in [20] and [32]. Finally, our candidate pruning mechanism, though serving a similar purpose as the techniques from [39] and [5], guarantees that no top-k result will be missed; thus our scheme is suitable for both exact as well as approximate top-k queries.

3 A New Top-k Query Processing Algorithm

In this section, we introduce our TBB (Threshold algorithm over Bucketized sorted lists with Bloom filter) scheme for top-k query processing. We begin by formulating the problem in Section 3.1. Next, we give an overview of our solution approach in Section 3.2, before developing our input depth estimation model in Section 3.3 and the candidate pruning mechanism in Section 3.4. For clarity, we start with a simple model with weighted sum, and attributes that are independent, identically, uniformly distributed in the range $[0, \alpha]$, and gradually generalize our solution. In Section 3.5, we extend our techniques to cope with non-uniform data distributions as well as correlated attributes. This is followed by the data structures and the query processing algorithm for our solution in Section 3.6. Finally, Section 3.7 discusses how our solution generalizes to top-k queries involving nested scoring functions and selection predicates.

Notation	Meaning	Default
D	Database of records	–
n	Number of records in D	1 million
m	Number of record attributes	20
A_j	The j-th record attribute	–
L_j	The sorted list on A_j	–
$[0, \alpha]$	Domain of record attributes	$[0, 1]$
Q	A user query	–
q	Number of query attributes	2
w_j	Weight of the j-th query attribute	1
r	A record in D	–
$S(r\|Q)$	Score of r w.r.t. Q	–
$\overline{S}(r\|Q)$	Upper-bound score of r w.r.t. Q	–
$\underline{S}(r\|Q)$	Lower-bound score of r w.r.t. Q	–
R	Result of Q	–
k	Number of required records in R	10
S_k	k-th highest record score in R	–

Table 1 Query Model

3.1 Problem Formulation

We have a database D consisting of n records. Each record contains m attributes with values falling uniformly in the range $[0, \alpha]$[1]. We also have m sorted lists L_j, each of which sorts the records in descending order on a different attribute A_j. Each list L_j contains an entry $\langle r.A_j, r\rangle$ for each record r in the database D, where $r.A_j$ denote the value of the j-th attribute of record r. A query Q selects any of q number of attributes from the overall number of m attributes in the database. Without loss of generality, we assume that attributes A_1, A_2, ..., A_q are chosen; we call the corresponding sorted lists, L_1, L_2, ..., L_q, the *query lists*. The score of a record $r \in D$ with respect to Q, $S(r|Q)$, is a monotonic aggregation function such that for any two records $r_1, r_2 \in D$, if $r_1.A_j \geq r_2.A_j$ for every $1 \leq j \leq q$, then $S(r_1|Q) \geq S(r_2|Q)$. The query result R is a ranked list of k records such that:

[1] We work with the range of $[0, \alpha]$, instead of $[0, 1]$, as a prelude to supporting different data distributions and weights for the attributes. This will become evident in Section 3.5.

– The records in R are ordered in decreasing scores, i.e., $R = [r_1, r_2, \ldots, r_k]$ where $S(r_i|Q) \geq S(r_j|Q)$ for $i < j$ and $r_i, r_j \in R$; and
– All other records in D do not have higher scores than any result record, i.e., $\forall r_i \in D - R, S(r_i|Q) \leq S_k$ where S_k is the score of the last result record in R.

3.2 Motivation for Our Solution Approach

Algorithm 1 NRA to retrieve the top-k results

```
1: Initialize sorted result R.
2: Set depth d = 1.

3: // Phase One ...
4: while d ≤ n do
5:     Set threshold τ = 0.
6:     for j = 1 to q do
7:         Let ⟨r.A_j, r⟩ be the d-th entry in L_j.
8:         if r is discovered for the first time then
9:             Create an entry in R for r.
10:            Set S̲(r|Q) = w_j × r.A_j.
11:        else r has appeared in some other query list
12:            Set S̲(r|Q)+=w_j × r.A_j.
13:        Set τ+=w_j × r.A_j.
14:    d+=1.
15:    if (|R| ≥ k) AND (S_k > τ) then
16:        Goto line 18.

17: // Phase Two ...
18: while d ≤ n do
19:    for j = 1 to q do
20:        Let ⟨r.A_j, r⟩ be the d-th entry in L_j.
21:        if r is discovered for the first time then
22:            Ignore ⟨r.A_j, r⟩.
23:        else r has appeared in some other query list
24:            Set S̲(r|Q)+=w_j × r.A_j.
25:    d+=1.
26:    if terminating conditions are met then
27:        Goto line 29.

28: // Return the top-k results
29: Remove from R the (k+1)-th entry downwards.
30: Return R.

31: Terminating conditions:
32: • ∀1 ≤ i_1 < i_2 ≤ k, S̲(R.r_{i_1}|Q) ≥ S̄(R.r_{i_2}|Q), and
33: • ∀i > k, S̲(R.r_k|Q) ≥ S̄(R.r_i|Q).
```

We begin by examining the behavior of the classical NRA algorithm from [14]. In the pseudo-code in Algorithm 1,

– $\underline{S}(r|Q)$ gives a lower bound on the score of r with respect to Q. $\underline{S}(r|Q) = \sum_{j=1}^{q} w_j \times \varphi_j$ where $\varphi_j = r.A_j$ if the entry $\langle r.A_j, r\rangle$ has been discovered in query list L_j; otherwise $\varphi_j = 0$.
– $\overline{S}(r|Q)$ gives an upper bound on the score of r with respect to Q. $\overline{S}(r|Q) = \sum_{j=1}^{q} w_j \times \gamma_j$ where $\gamma_j = r.A_j$ if the entry $\langle r.A_j, r\rangle$ has been discovered in query list L_j; otherwise γ_j corresponds to the last attribute value read from L_j.

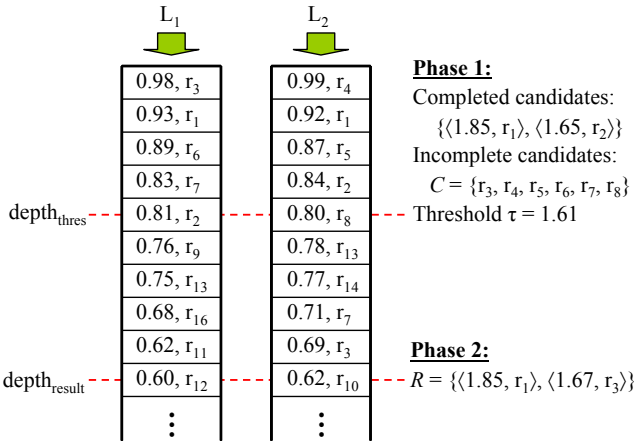


Fig. 1 NRA Execution for Top-2 Objects from L_1 and L_2

The algorithm maintains a sorted list of candidate results, ordered in descending $\underline{S}(r|Q)$ scores. A candidate r is said to be *completed* if all of its query attributes have been discovered, so $\underline{S}(r|Q) = \overline{S}(r|Q)$; otherwise r is *incomplete*.

Since γ_j changes every time an entry is popped from L_j, we do not update all the $\overline{S}(r|Q)$ values eagerly. Instead, a given $\overline{S}(r|Q)$ is computed on-demand when it is needed to test the terminating conditions, by adding to $\underline{S}(r|Q)$ the γ_j value for those lists L_j in which r has yet to be discovered.

Figure 1 exemplifies NRA's execution for a top-2 query on two sorted lists. The algorithm scans all the query lists to the same depth. At $depth_{thres}$, there are two completed records r_1 and r_2 with scores $S(r_1|Q) = 1.85$ and $S(r_2|Q) = 1.65$, and the threshold $\tau = 0.81 + 0.80 = 1.61$. Since τ forms an upper bound on the score of any new record down the lists, none of those records can outscore r_1 or r_2, so phase one of NRA ends. At this time, R comprises the tentative answer $\{\langle 1.85, r_1\rangle, \langle 1.65, r_2\rangle\}$, as well as incomplete candidates $C = \{r_3, r_4, r_5, r_6, r_7, r_8\}$. In the next phase, NRA scans further down the lists to $depth_{result}$, where r_1 and r_3 (with $S(r_3|Q) = 1.67$) are confirmed to have the highest scores while the remaining candidates in C are found to be non-viable.

Although NRA is guaranteed to locate the top-k results correctly, it has the following inefficiencies:

– NRA scans one entry from each query list at a time, which generates random I/Os and is slow. If we could determine $depth_{thres}$ and $depth_{result}$ right from the start, we could fetch all $depth_{thres}$ entries at once from each list in phase one, then the next $(depth_{result} - depth_{thres})$ entries at once from the

Notation	Meaning
p	Number of buckets to split each L_j into
b	Number of records per bucket
B	A bucket of records
$LB(B)$	Lowest attribute value in B
$UB(B)$	Highest attribute value in B
$B_j(r)$	First bucket in L_j that contains record r
FP_β	False positive rate of bucket β in each L_j

Table 2 Parameters for Bucket Design

lists in phase two. This way, the execution would be sped up through sequential or batched I/O operations.

– Phase two of NRA evaluates some candidates (like r_4, r_5, r_6, r_8 in Figure 1) that eventually are found to be non-viable because they do not appear within the first $depth_{result}$ entries in one or more of the lists. Maintaining the lower- and upper-bound scores of those candidates during the second phase is a waste of computation cycles and memory. If the upper-bound scores could be estimated, such candidates might be ruled out early to improve efficiency.

We will show how to derive $depth_{thres}$ and $depth_{result}$ next, followed by the estimation of upper-bound scores in Section 3.4.

3.3 Estimation of $depth_{thres}$ and $depth_{result}$

We partition each sorted list L_j into $p = \frac{n}{b}$ buckets with b entries each; the buckets within each sorted list are numbered serially from 0 onwards. For simplicity, we assume that n is a multiple of b so that p is a positive integer. The buckets in each list are numbered $0, 1, \ldots, p-1$, with bucket 0 at the head, and bucket $p-1$ at the tail-end. We show in the following theorem how to fix the bucket size and determine $depth_{thres}$. Let $B_j(r)$ denote the bucket in the query list L_j that contains the record r and let $b_{j,r}$ be the bucket number of $B_j(r)$.

Theorem 1 *Suppose that for every $j \in [1, m]$, for every $r \in D$, $b_{j,r}$ is uniformly distributed over $[0, p-1]$. If we set the bucket size $b = k^{1/q} \cdot n^{(q-1)/q}$, then it is expected that there are k records with completed scores after NRA phase one fetches the first bucket for each queried attribute. Namely, $depth_{thres} = k^{1/q} \cdot n^{(q-1)/q}$.*

PROOF(informal) Consider any record r appearing in bucket 0 of L_1. The probability that r also appears in all the first buckets of $L_2, \ldots, L_q$ is $\frac{1}{p^{q-1}}$. Since there are b records per bucket, the expected number of such records is $\frac{b}{p^{q-1}}$. Since by definition $bp = n$,

$$p = \frac{n}{k^{1/q} \cdot n^{(q-1)/q}} = \left(\frac{n}{k}\right)^{\frac{1}{q}} \tag{1}$$

we have $\frac{b}{p^{q-1}} = k$. In other words, it is expected to discover k records with completed scores by setting

$$depth_{thres} = b = k^{1/q} \cdot n^{(q-1)/q} \tag{2}$$

□

Obviously, we want to have $p \geq 2$; otherwise, with only one bucket per sorted list, our algorithm would be scanning the entire query lists which is inefficient. This implies that our algorithm is meaningful only for $n \geq k \cdot 2^q$.

Next, let $LB(B)$ and $UB(B)$ denote the lowest and highest attribute values in bucket B respectively. Moreover, let $BC(r)$ denote the sum of the serial number of the buckets in the query lists that contain r; i.e., $BC(r) = \sum_{j=1}^{q} b_{j,r}$. If we assume $b = 5$ in the example in Figure 1, $b_{1,r_7} = 0$ and $b_{2,r_7} = 1$, so $BC(r_7) = 0+1 = 1$. We show in Theorem 2 how to establish $depth_{result}$.

Theorem 2 *Suppose that for every $j \in [1, q]$, $r.A_j$ is distributed uniformly in $[0, \alpha]$. Then, for any L_j, a record r is not expected not to be qualified if $BC(r) \geq q$ under the same bucketization setting as in Theorem 1. In other words, $depth_{result} = q \cdot k^{1/q} \cdot n^{(q-1)/q}$.*

PROOF(informal) We prove the theorem by showing that in average there exit k records with scores larger than r's.

Note that for $0 \leq i < p$, all the attributes in bucket i of L_j is bounded by $\frac{p-i-1}{p}\alpha$ and $\frac{p-i}{p}\alpha$. Thus we have

$$\begin{aligned} S(r|Q) &\leq \sum_j \frac{p - b_{j,r}}{p}\alpha = \frac{\alpha}{p}\sum_j (p - b_{j,r}) \\ &= \frac{\alpha}{p}(pq - BC(r)) \leq \frac{p-1}{p} q\alpha \end{aligned}$$

The threshold τ is at the end of bucket 0 in all the query lists after phase one, so $\tau = \frac{p-1}{p} q\alpha$ too. Since it is expected that we already have k completed records scoring above τ because all of their attribute values appear in the first bucket of the query lists, r cannot qualify for the top-k answer and so can be pruned.

Therefore, it suffices for phase two to scan to the end of bucket $q-1$ of the query lists; any candidate record that still has undiscovered attribute values at that point can no longer be viable. With b entries per bucket, we have:

$$depth_{result} = q \cdot k^{1/q} \cdot n^{(q-1)/q} \tag{3}$$

□

q	$ncand_{phase1}$	$ncand_{ideal}$
2	6,314	30
3	64,623	100
4	224,927	350
5	499,990	1260
6	880,670	4620
7	1,000,000	17,160
8	1,000,000	64,350

Table 3 The number of candidates found in phase one ($ncand_{phase1}$), versus the ideal number of candidates for phase two ($ncand_{ideal}$)

3.4 Estimation of Upper-Bound Scores for Candidate Records

With q query lists, the number of bucket combinations that allow a record r to have $BC(r) \leq \beta$, for some $0 \leq \beta \leq p(q-1)$, is

$$\#\text{ bucket combinations}(BC(r) \leq \beta) = \prod_{j=1}^{q} \frac{\beta + j}{j}$$

Intuitively, the above formula calculates, in the hyperspace formed by the q query attributes, the volume bounded by the origin and the hyperplane that is anchored at $\langle \beta, 0, \ldots, 0\rangle$, $\langle 0, \beta, \ldots, 0\rangle$, …, $\langle 0, 0, \ldots, \beta\rangle$.

Since every bucket combination has the same likelihood of $\frac{1}{p^q}$ and the database contains n records, the expected number of records r for which $BC(r) \leq \beta$ is

$$\#\text{ records}(BC(r) \leq \beta) = \frac{n}{p^q} \cdot \prod_{j=1}^{q} \frac{\beta + j}{j}$$

As explained in Section 3.3, a record r cannot qualify for the top-k answer if $BC(r) \geq q$. Therefore ideally the number of candidate records that should be evaluated in phase two of the algorithm is $ncand_{ideal} = \#\text{ records}(BC(r) \leq q-1) = \frac{n}{p^q} \cdot \prod_{j=1}^{q} \frac{q-1+j}{j}$. In contrast, phase one of the algorithm retrieves $q \cdot b$ records, among which only k scores above the threshold τ, leaving $ncand_{phase1} = q \cdot b - k$ candidates. Table 3 illustrates the difference between $ncand_{phase1}$ and $ncand_{ideal}$, for $n = 1,000,000$ and $k = 10$. Evidently, the difference $(ncand_{phase1} - ncand_{ideal})$ widens rapidly as q increases. A filtering mechanism that is able to prune most of the non-viable candidate records, without incurring random I/Os in retrieving their attribute values, could therefore give a significant performance boost to the algorithm, at the expense of some memory overhead (which we will quantify through experiments in Section 4.

We base our filtering mechanism on the Bloom filter. As proposed in [6], a Bloom filter is designed to support membership checks on a set B of b key values, $B = \{r_1, r_2, \ldots, r_b\}$. To construct a Bloom filter with ι bits, we choose ν independent hash functions $h_1, h_2, \ldots, h_\nu$, each with a range of $[1, \iota]$. For each value $r_i \in B$, the filter bits at positions $h_1(r_i), h_2(r_i), \ldots, h_\nu(r_i)$ are set to 1. To check whether a given r is in B, we examine the bits at $h_1(r), h_2(r), \ldots, h_\nu(r)$. If any of the bits is 0, r cannot possibly be in B; otherwise there is a high probability that r is in B. In other words, the Bloom filter admits controlled false positive rates but no false negatives. The false positive rate is

$$FP = \left(1 - \left(1 - \frac{1}{\iota}\right)^{\nu b}\right)^{\nu} \approx \left(1 - e^{-\nu b/\iota}\right)^{\nu} \quad (4)$$

Mathematically, FP is minimized for $\nu = (\iota \times ln2)/b$. Since ν must be an integer, we will use $\nu = \lfloor(\iota \times ln2)/b\rfloor$. Given the value of b and the target FP rate, we can thus set ν and ι accordingly.

For each bucket B, we maintain a triplet $\langle LB(B), UB(B), BF(B)\rangle$ where $LB(B)$, $UB(B)$, and $BF(B)$ are the smallest attribute value, the largest attribute value, and the Bloom filter on the record identifiers in bucket B, respectively. At runtime, the Bloom filters (which are much more compact than the sorted lists) are loaded into memory, and used for filtering the candidate records C that are found in phase one of the threshold algorithm. Let $B_j(r)$ be the first bucket in sorted list L_j that, according to the Bloom filter on the bucket $BF(B_j(r))$, *might* contain r. The probability of $B_j(r) = \beta$ is

$$Prob(B_j(r) = \beta) =$$

$$\begin{cases} \frac{1}{p} & \text{if } \beta = 0 \\ \left(\frac{1}{p} + \frac{p-\beta-1}{p} FP_\beta\right) \prod_{h=1}^{\beta-1} (1 - FP_h) & \text{if } 1 \leq \beta < p \\ 0 & \text{otherwise} \end{cases}$$

where FP_β is the false positive rate of the Bloom filter for the β-th bucket of the sorted lists. There is no false positive component associated with $\beta = 0$ because bucket 0 of every query list is always loaded into memory, so its contents can be examined directly.

For any record r, we derive an upper bound for its score from the Bloom filters,

$$S(r|Q) \leq \sum_{j=1}^{q} UB(B_j(r))$$

Any record r for which $\sum_{j=1}^{q} UB(B_j(r)) < \tau = \frac{p-1}{p} q\alpha$ can be eliminated.

The above filtering mechanism can be optimized in the following ways.

– If a record r scores less than $\frac{p-q}{p}\alpha$ for even one query list, r will not be able to meet the cut-off threshold τ even if it gets the maximum α from each of the remaining $q-1$ lists. Therefore we only need to construct Bloom filters for the leading buckets B in each sorted list for which $UB(B) \geq \frac{p-q}{p}\alpha$, not for the trailing buckets. Once the algorithm determines that a record r is not found within those leading buckets of a query list, r can be disqualified immediately.
– In phase two, as the algorithm evaluates the candidates in C, it is likely to find among them records that accumulate sufficiently high scores to qualify for R. As S_k (the k-th largest record score) rises in the process, further records r in C can then be discarded if their upper-bound score as determined from the Bloom filters is below S_k, i.e., $\sum_{j=1}^{q} UB(B_j(r)) < S_k$.
– The algorithm always retrieves bucket 0 of the query lists in order to formulate the initial R. A Bloom filter on bucket 0 serves no purpose since the algorithm can check the bucket content directly; we thus build the Bloom filter from bucket 1 downwards.

3.5 Handling Practical Attribute Distributions

In formulating our scheme so far, we have made simplifying assumptions such as the attributes are identically and independently distributed, and the attribute values are uniformly distributed in $[0, \alpha]$. We now consider how to generalize the scheme beyond those assumptions so that it is capable of supporting realistic applications.

Non-Identical and Non-Uniform Attribute Distributions

In some applications, the query attributes may not be identically distributed. Rather, each attribute A_j could have a different range $[\underline{\alpha}_j, \overline{\alpha}_j]$, and the attribute values may not be distributed uniformly within the range. Even where the attributes share a common domain $[\underline{\alpha}, \overline{\alpha}]$, the query could associate different weights w_j with the attributes.

Figure 2 depicts the extension of our proposed scheme to cope with non-identical attribute distributions. As before, we partition each attribute list into p buckets with b entries each, but allow the buckets to cover varying intervals of attribute values. (This bucket organization is analogous to an equi-depth histogram.) Denoting the smallest and largest attribute values in bucket 0 of L_j by θ_j and $\overline{\alpha}_j$, and the probability that a record falls within $[\theta_j, \overline{\alpha}_j]$ by $P(A_j \geq \theta_j)$,

$$P(A_j \geq \theta_j) = \frac{b}{n} = \frac{1}{p} \tag{5}$$

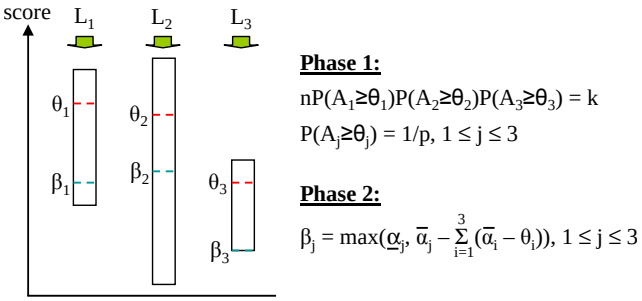


Fig. 2 Non-Identical Attribute Distributions

By setting p as in Formula 1, phase one of the algorithm is still expected to yield k records with completed scores after collecting the entries in bucket 0 of the query lists. Again, S_k denotes the k-th highest record score ($S_k = \underline{S}(R.r_k|Q)$) thus far; $S_k \geq \tau = \sum_{j=1}^{q} w_j \times \theta_j$. In phase two, rather than uniformly fetching $depth_{result}$ (as defined in Formula 3) entries from every list, we now fetch from each query list L_j the remaining buckets B that satisfy $UB(B) \geq \beta_j$, where

$$\beta_j = max\left(\underline{\alpha}_j,\ \overline{\alpha}_j - \sum_{i=1}^{q} \overline{\alpha}_i + S_k\right) \tag{6}$$

Other than these modifications for the expected input depths, the estimation of upper-bound scores as presented in Section 3.4 still applies, and it allows us to prove that any record r appearing below β_j in some query list L_j cannot possibly qualify as a top-k result. This extended scheme works with any arbitrary attribute distribution. A skewed distribution, for example, would just lead to varying interval widths $UB(B) - LB(B)$ across buckets.

Attributes with Positive Linear Correlation

It is common for some of the attributes in the data to be correlated. If two attributes are positively correlated, a record that ranks highly with respect to one attribute is likely to have a relatively large value for the other attribute. With negatively correlated attributes, a record that appears near the front of one attribute's sorted list is likely to be positioned towards the end of the other's sorted list. For instance, the sorted lists for synonyms like 'jurist' and 'judge' in a text corpus will be positively correlated, whereas opposite measures such as 'distance' and 'similarity' are likely to have negatively correlated sorted lists.

Where query attributes are positively correlated, and/or the first buckets of the query lists have disproportionately higher scores than the buckets that follow, the algorithm is likely to discover more than k

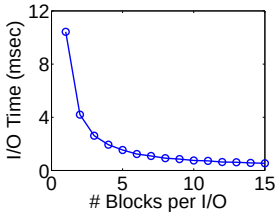


Fig. 3 Average I/O Cost with Sequential Reads

records with completed scores after phase one. To illustrate, consider the extreme case of a query for the top $k = 10$ records with respect to $q = 5$ attributes, all of which are set to the same value within each record. In essence, the query involves only one *independent* attribute, and it suffices to scan only the first $k = 10$ entries from each query list to compose the top-k record scores. This corresponds to setting $q = 1$ in Formula 2. In contrast, with $n = 1,000,000$ and $q = 5$, we get $b = 10^{1/5} \cdot 1000000^{4/5} = 100,000$; this causes the system to fetch 10,000 times more records than is necessary!

The above example highlights that, in Formulas 1 and 2, the setting of q should discount query attributes that are positively correlated. In order to do that, the system needs to track the correlation coefficient ρ_{ij} between every pair of attributes A_i and A_j. Since we are interested in the top-k results, it is not necessary to involve all the records in deriving the ρ's. Instead, we select only records that rank high for at least one attribute (say, the top-1,000 entries in each attribute list), so as to lower the computation overheads. Given a query Q, we repeatedly drop one of the query attributes A_j that satisfies both of the following conditions: (a) A_j is positively correlated with at least one other query attribute A_i, i.e., $\rho_{ij} > 0$; and (b) ρ_{ij} is significant at some pre-determined confidence level (e.g., 95%). At the end of the procedure, the remaining query attributes either are negatively correlated or there is no significant correlation among them. We now set q to the number of remaining attributes, to derive p and b with Formulas 1 and 2. With these p and b settings, we then perform query processing on the sorted list of all the query attributes, i.e., including the positively correlated attributes.

The above procedure is designed to discount query attributes that are determined with high confidence to be positively correlated. Even after applying the procedure, it is possible to still have weaker positive correlation among the remaining query attributes that causes our scheme to overestimate b and, hence, the number of blocks to scan in phase one. As an additional safeguard, we need to cap b without losing too much of the advantage of sequential I/Os. In general, the average I/O cost for fetching a block of sorted list entries from the disk trends down as a larger number of contiguous blocks are fetched in one I/O operation, because the disk seek and rotational delays are amortized over the blocks that are retrieved sequentially. However, the incremental gain becomes marginal beyond a certain point, say x blocks. For our test system, this point occurs at around $x = 6$ blocks as Figure 3 shows. (With 1-Kbyte blocks and 8 bytes per sorted list entry, as described in Section 4.1, 6 blocks corresponds to 750 sorted list entries.)

Based on this disk characteristic, we modify our threshold algorithm to cycle through the query lists in phase one, fetching up to x successive blocks from each list in turn, until the target $depth_{thres}$ is reached or at least k completed record scores are found, whichever occurs earlier. Likewise, phase two of the algorithm retrieves up to x blocks from each query list in turn, until the query answer is confirmed. This safeguard kicks in only in exceptional situations where the combination of a large k, a large q and moderate positive correlations cause $depth_{thres}$ and $depth_{result}$ to be elevated. In our experiments, the $depth_{thres}$ and $depth_{result}$ settings given by our model provide tighter limits in almost all cases.

Attributes with Negative Linear Correlation

In contrast, negative correlation among the query attributes would lower the initial S_k. Indeed, phase one of the algorithm may not even be able to locate k completed records from the first bucket of the query attributes. The algorithm will then need to delve deeper into the query lists in order to produce R. In particular, some valid result records may now score below $\frac{p-q}{p}\alpha$ on some query list. If such a circumstance is deemed likely, the default countermeasure is to extend the Bloom filters to all the blocks in the sorted lists, and the candidate filtering mechanism in Section 3.4 can still be effective. Alternatively, to conserve memory space, we may use progressively fewer Bloom filter bits per record. This permits a slightly elevated false positive rate for those buckets toward the end of the sorted lists, but it should not adversely affect performance because the attribute values in those buckets are low anyway.

NULL Attribute Values

In some applications, a record may not specify a value for every attribute. A classic example is the text search engine [49], where the records correspond to documents and the attributes correspond to search terms. Since a document typically contains only a small subset of the possible search terms, for efficiency reasons the document is omitted from the sorted list of those search

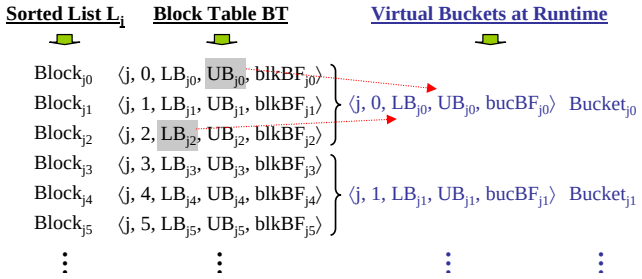


Fig. 4 Dynamic Block-to-Bucket Mapping
(In this example, each bucket is 3 disk blocks in size)

terms that it does not contain. Consequently, each sorted list indexes a *subset* of the documents rather than the entire document collection, and the sorted lists may vary in length.

Since unspecified attributes do not contribute to the score of a record, we *conceptually* augment the sorted list L_j of attribute A_j with an entry $\langle 0, r\rangle$ for every record r that does not specify a value for A_j, so that L_j again indexes the full set of n records. This allows us to apply Formulas 1 and 2 to determine settings for p and b, followed by Formula 6 to decide how deep to scan each query list. Once the query processing algorithm reaches the end of a sorted list L_j, any incomplete candidate record that has not appeared in L_j can assume zero contribution from A_j.

3.6 Algorithm Implementation

We now put together the algorithms for our proposed solution below.

System Configuration

According to Formulas 1 and 2, the settings for p and b are dependent on query-specific parameters k and q. It is thus not possible to partition the sorted lists into buckets and generate their associated Bloom filters in advance to cater for all possible queries. Instead, we store each sorted list in contiguous disk blocks, and construct for each disk block a Bloom filter with the specified false positive rate FP. The details of every block B_{ji} are captured as a tuple $\langle j,\ i,\ LB(B_{ji}),\ UB(B_{ji}),\ blkBF_{ji}\rangle$ in a 'Block Table' BT, where j is the identifier of the sorted list L_j, i is the block offset within L_j, $LB(B_{ji})$ and $UB(B_{ji})$ are the smallest and largest attribute values within B_{ji}, and $blkBF_{ji}$ is the Bloom filter for the record identifiers in B_{ji}. At runtime, after the bucket boundaries have been determined for a given query, the blocks in each query list are mapped to the buckets dynamically. Figure 4 illustrates the mapping, with 3 blocks to a bucket. Thus, checking the bucket Bloom filter $bucBF$ (e.g. bucBF_{j1}) for a record identifier entails probing the Bloom filter $blkBF$ of its underlying disk blocks ($blkBF_{j3}$, $blkBF_{j4}$ and $blkBF_{j5}$). This storage scheme incurs overheads in creating one tuple per block in BT (rather than one BT tuple per multi-block bucket), and in examining multiple block-level Bloom filters for each bucket probe. In return, we gain the flexibility of catering to ad-hoc queries with arbitrary k and q requirements.

Algorithm 2 Block Table Construction for the Sorted Lists

Function: CreateBF(B, b, FP)
...
Return bloom filter FP.

Function: Construction(n, k_{max}, q_{max}, L_j, FP)
1: Initialize the Block Table BT.
2: Let b be the number of sorted list entries per disk block.
3: Set $p = \lceil \frac{n}{b} \rceil$.
4: Store the first b entries of L_j in block B_{j0}.
5: Append $\langle j, 0, LB(B_{j0}), UB(B_{j0}), NULL\rangle$ to BT.
6: **for** $i = 1$ to $p - 1$ **do**
7: Store the next b entries of L_j in block B_{ji}.
8: Let $blkBF_{ji}$ = CreateBF(B_{ji}, b, FP).
9: Append $\langle j, i, LB(B_{ji}), UB(B_{ji}), blkBF_{ji}\rangle$ to BT.

Algorithm 2 summarizes the configuration algorithm. Here, we are creating Bloom filters for all the blocks in each sorted list. If the applications are able to impose upper limits on the query answer size (k_{max}) and the number of query attributes (q_{max}), we can calculate with Formula 3 the maximum depth to which the sorted lists will ever be scanned, and create Bloom filters only up to that depth. For example, queries in a text search engine typically do not exceed 20 query terms, and most users do not look beyond the first page of 10 result entries [36]. If it is not possible to fix specific values for k_{max} and q_{max}, all the blocks in each sorted list will get a Bloom filter by default.

Our approach of embeding a Bloom filter on the entries within each disk block of a sorted list has the advantage of facilitating data updates. Whenever there are changes to the entries assigned to any disk block, the Bloom filter within it can be refreshed at the same time, and written back to disk without incurring extra I/O operations.

Query Processing

The query processor and its associated functions are presented in Algorithm 3. In phase one of query processing, lines 36 and 37 are intended to handle situations where the tentative answer R contains fewer than

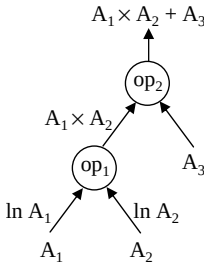


Fig. 5 Query Plan for Nested Top-k Operations

k records that score above the threshold τ after processing the first bucket from all the query lists. This could be caused by a number of factors; e.g., the query attributes are negatively correlated as explained previously. Under such circumstances, the algorithm needs to examine additional blocks of records from the query lists to ensure that all candidate records are located.

Similarly, lines 52 to 54 in Algorithm 3 catch exceptions where, after processing the buckets B_{ji} that satisfy $UB(B_{ji}) \geq \beta_j$, there is still no total ordering among the top-k candidates, or there are still candidates with incomplete scores that cannot be disqualified. These exceptions could be handled by scanning deeper down the query lists. However, as the number of candidates in question is expected to be low, we opt to retrieve their attribute values directly to complete their aggregate scores.

3.7 Extensions to Our Query Model

Up to this point, we have focused on scoring functions that are weighted sums of the query attributes. Our solution extends trivially to linear aggregation functions of the form $S(r|Q) = \sum_{j=1}^{q} w_j \times f(r.A_j)$ where f is a monotonic increasing function. In this section, we discuss how our solution can apply to queries that involve nested top-k operations, and queries that have selection predicates on the query lists.

Queries with Nested Top-k Operations

To simplify our discussion, we center on the scoring function $S(r|Q) = A_1 \times A_2 + A_3$; extension to scoring functions that involve more products and additions of attributes is straightforward.

Instead of a single operation, the query requires two top-k operators as illustrated in Figure 5. The first operator op_1 implements the scoring function $lnA_1 + lnA_2$, whose output is dynamically mapped to $e^{lnA_1+lnA_2} = A_1 \times A_2$ for the second operator op_2 that scores on $(A_1 \times A_2) + A_3$.

Algorithm 3 Query Processing with Bucketized Sorted Lists

Function: ProbeBF(r, BF)
...
Return TRUE/FALSE.

Function: UBScore(r)
1: Set score $S = 0$.
2: **for** $j = 1$ to q **do**
3: **if** r has appeared in B_{j0} **then**
4: Set S += $UB(B_{j0})$.
5: Continue with the next j iteration.
6: **for** $i = 1$ to $p - 1$ **do**
7: **if** ProbeBF(r, $bucBF_{ji}$) is TRUE **then**
8: Set S += $UB(B_{ji})$.
9: Break out of for-loop i.
10: Return S.

Function: Phase2Done()
11: Terminating conditions:
12: • $\forall 1 \leq i_1 < i_2 \leq k, \underline{S}(R.r_{i_1}|Q) \geq \overline{S}(R.r_{i_2}|Q)$, and
13: • $\forall i > k, \underline{S}(R.r_k|Q) \geq \overline{S}(R.r_i|Q)$.

14: **if** the terminating conditions are met **then**
15: Return TRUE;
16: **else**
17: Return FALSE;

Function: Query(k, q)
18: // Compose the buckets dynamically
19: Calculate b with Formula 2.
20: Round b up to the nearest multiple of blocks of records.
21: Set $p = \lceil \frac{n}{b} \rceil$.
22: **for** $j = 1$ to q **do**
23: **for** $i = 0$ to $p - 1$ **do**
24: Map $\langle j, i, LB(B_{ji}), UB(B_{ji}), bucBF_{ji} \rangle$ from the underlying blocks with information in BT.

25: // Phase One ...
26: Set $\tau = 0$.
27: **for** $j = 1$ to q **do**
28: Fetch bucket B_{j0} of L_j.
29: **for all** $\langle r.A_j, r \rangle$ in B_{j0} **do**
30: **if** r is discovered for the first time **then**
31: Create an entry in R for r.
32: Set $\underline{S}(r|Q) = w_j \times r.A_j$.
33: **else** r has appeared in some other query list
34: Set $\underline{S}(r|Q)$ += $w_j \times r.A_j$.
35: Set τ += $w_j \times LB(B_{j0})$.
36: **if** $S_k < \tau$ **then**
37: Repeat lines 27–35, replacing B_{j0} with the next disk block of L_j.

38: // Prune non-viable candidates
39: **for all** incomplete r in R **do**
40: **if** UBScore(r) $< S_k$ **then**
41: Remove r from R.

```
// Phase Two ...
for j = 1 to q do
    Calculate β_j with Formula 6.
    for all bucket B_ji such that UB(B_ji) ≥ β_j do
        Fetch bucket B_ji of L_j.
        for all ⟨r.A_j, r⟩ in B_ji do
            if r is discovered for the first time then
                Ignore ⟨r.A_j, r⟩.
            else r has appeared in another query list
                Set S(r|Q) += w_j × r.A_j.
while Phase2Done() is FALSE do
    Fetch the attribute values of a candidate record r that violates the terminating conditions.
    Complete r's aggregate score S(r|Q).

// Return the top-k results
Remove from R the (k+1)-th entry downwards.
Return R.
```

The techniques that we presented earlier can be applied to the query plan, as follows:

- Working backwards from the result output, our scheme for estimating $depth_{thres}$ and $depth_{result}$ is applied to the second operator op_2 to determine the required depths on $(A_1 \times A_2)$ and on A_3. The $depth_{thres}$ and $depth_{result}$ on $A_1 \times A_2$ are in turn translated to the input depths for the first top-k operator op_1, again using our scheme to estimate the input depths. These depth estimates enable the server to fetch the required entries from the A_1, A_2 and A_3 lists efficiently, through sequential or batched I/Os as before.
- In the course of executing the query, non-viable candidates can still be pruned with the bucketized Bloom filters on A_1, A_2 and A_3. The exception is the intermediate list for $(A_1 \times A_2)$ which has no pregenerated Bloom filters and hampers the pruning of candidates from A_3. Note that candidates from $(A_1 \times A_2)$ can still be pruned with the Bloom filters on A_3.

Top-k Queries with Selection Predicates

In the general form, a top-k query may have selection predicates on one or more of the query lists. We consider separately the cases where the selection is on the scoring attribute of a query list, and where the selection and scoring attributes are different.

- In the first case, the selected entries occupy a continuous range in the query list. We simply apply the estimated input depths $depth_{thres}$ and $depth_{result}$ with respect to the selected sublist.
- In the second case, the selected entries are scattered across the query list, which is sorted on the scoring attribute. Denoting the selectivity factor of the range selection as sf, and assuming that the selection attribute is independent of the scoring attribute, we need to scale the estimated depths by a factor of $\frac{1}{sf}$, to become $depth_{thres}/sf$ and $depth_{result}/sf$ respectively.

4 Empirical Evaluation

In this section, we experimentally evaluate the performance of our proposed TBB scheme for top-k query processing. The questions that we are seeking answers to include:

- What are the relative strengths and weaknesses of TA versus NRA, in processing workloads with different data distributions?
- How accurately does our model (particularly Formulas 2, 3 and 6) predict $depth_{thres}$ and $depth_{result}$, which are pivotal in sequentially fetching just the right number of disk blocks for each query list?
- How effective is our bucket organization with its associated Bloom filters in weeding out non-viable candidate records, and in ordering the viable ones?
- What is the overall performance impact of our proposed TBB scheme, and how adaptable is it to different data distributions?

After describing the experiment set-up, we present the experiment results, followed by a summary of the answers to the above questions obtained from our empirical study. We continue to follow the notation and default parameter settings in Tables 1 and 2.

4.1 Experiment Set-Up

Algorithms: We implemented TA, NRA and our TBB scheme for the experiment study. Our implementation incorporated the best TA techniques (from [2]) and the state-of-the-art NRA techniques (from [25,26]). We also include a variant of NRA, denoted by bNRA, that employs prefetching to reduce seek delays in the disk; specifically, each disk block request causes 6 consecutive blocks to be fetched into the I/O cache, based on the disk characteristics in Figure 3 and the associated discussion in Section 3.5.

Datasets: We run the competing algorithms on four datasets. Three of the datasets are synthetic, containing respectively independent, positively correlated, and negatively correlated attributes, while the fourth dataset is derived from a TREC benchmark [40]. We will elaborate on the datasets as they are used in the experiments that follow.

Hardware: Our test system is a Redhat Linux server equipped with dual Intel Xeon 3GHz CPUs, 4GB RAM and a Seagate ST973401KC 73GB hard disk. The disk is formatted with 1-Kbyte blocks, the default in Linux. With 4-bytes each for the attribute value and the record identifier, every sorted list entry $\langle r.A_j, r\rangle$ occupies 8 bytes, and 125 entries can fit into each disk block. Our implementation fetches blocks of sorted list entries from the disk as they are required by the query processing algorithm. Only the Bloom filters are loaded in advance and remain cached in main memory.

Performance metrics: The primary performance metrics are: (i) The number of candidate records that need to be examined in phase two of query processing; this quantifies the effectiveness of our bucket organization in filtering out non-viable candidates. (ii) The depth to which the query lists are scanned; this measures the accuracy of our sequential block fetching mechanism. (iii) The overall query processing time, which includes the overhead of mapping dynamically the Bloom filter in the disk blocks to filters for the virtual buckets, and the computation overhead as well as I/O time for executing the queries. (iv) The memory requirement of the algorithms.

4.2 Attributes with Independent Distributions

We begin by studying how the various algorithms behave with independent attributes. The dataset comprises one million records with 20 attributes. Each attribute value is generated randomly from a uniform distribution over the range $[0, 1]$. Queries are composed from randomly selected attributes, and all the attributes are weighted equally (i.e., $\forall 1 \leq j \leq q$, $w_j = 1$). Each experiment result is averaged over 1000 queries and verified to be statistically significant. The results are summarized in Figure 6. For the timing measurements in Figure 6(c), the dotted line for each algorithm represents its I/O time, whereas the solid line corresponds to the total processing time.

Figure 6(a) shows that, at $q = 2$ and $k = 10$, TA finds an average of 4196 candidate records at the end of phase one. Fetching each of the candidate records generates one random I/O. At roughly 20 msec per random I/O, the I/O cost of 84 seconds makes up almost the entire processing time of TA. As k increases to 50, the number of such candidate records rises to 11606, pushing the processing time up to 234 seconds in Figure 6(c).

NRA identifies just as many candidate records as TA, as shown in Figure 6(a). Unlike TA, NRA continues to poll the sorted lists in phase two of query processing, until all the candidate records are either disqualified or completely ordered. According to Figure 6(b), this entails scanning 0.31% of the sorted lists after phase one, to 0.44% after phase two. The number of random I/Os generated to fetch blocks of the sorted lists is substantially lower than those that TA expends to fetch individual records, thus explaining the superior performance of NRA over TA here. The turnaround time of NRA is consistently around 2% that of TA in this experiment.

Scheme	# Cand	Depth	Time	Memory
TA	4,196	0.31%	84.9 sec	4 MB
NRA	4,297	0.44%	1.69 sec	5 MB
bNRA	4,297	0.44%	0.88 sec	11.7 MB
TBB-1%	138	0.44%	0.72 sec	41 MB
TBB-10%	1,011	0.44%	0.71 sec	28 MB
TBB-20%	1,806	0.44%	0.81 sec	25 MB
TBB-30%	2,448	0.44%	0.92 sec	23 MB
TBB-40%	2,989	0.44%	0.98 sec	21 MB

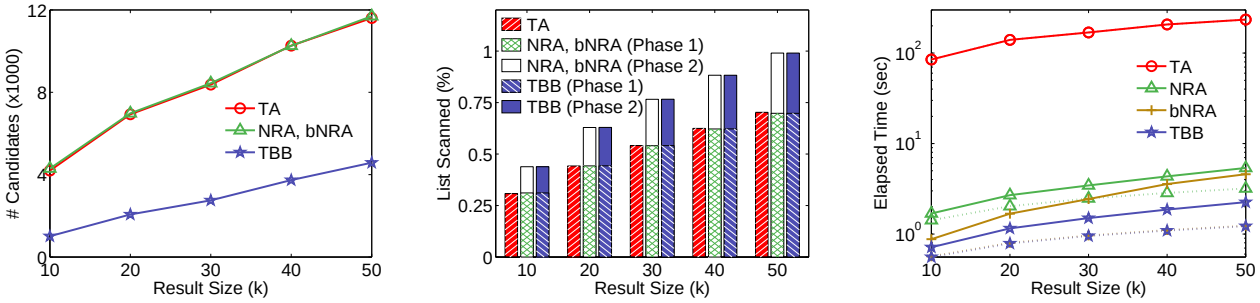
Table 4 Varying FP with $q = 2$ and $k = 10$

For bNRA, the number of candidate records and scan depths are the same as those of NRA. The only difference is that bNRA has lower I/O costs, due to its disk prefetching operations. This enables it to shave 15% off the processing time of NRA.

Turning our attention to TBB, Figure 6(b) shows that, at FP=10%, TBB scans the sorted lists only marginally deeper than NRA in phases one and two. This attests to the accuracy of the estimates yielded by our model, and is instrumental in reducing the I/O cost of TBB through sequential or batched block fetches. Moreover, our bucket organization enables TBB to examine less than 40% as many candidate records as NRA, thus lowering the computation cost. These two factors combined to make TBB the fastest algorithm, achieving speed-up of around 104, 2.4 and 2 times over TA, NRA and bNRA, respectively.

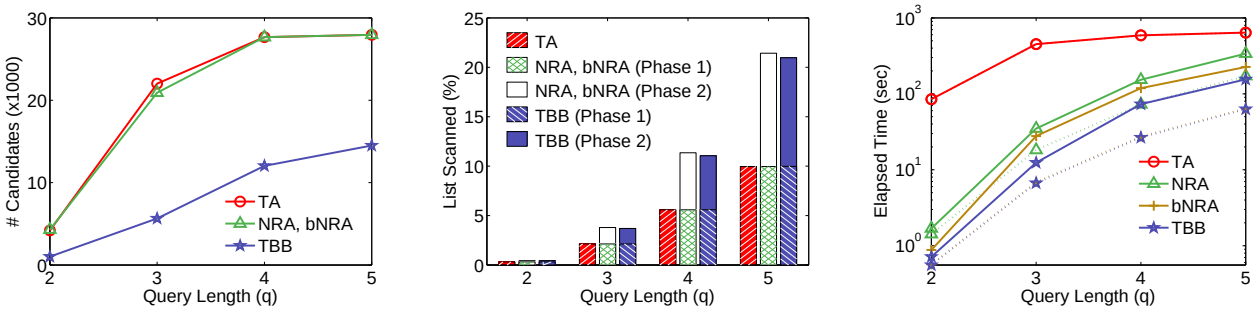
Figure 7 presents the experiment results with varying number of query attributes (query length) and k fixed at 10. Interestingly, NRA scans deeper into the sorted lists than TBB. This is necessary in order to resolve the huge number of candidate records found in phase one. For example, at $q = 5$, NRA is saddled with 27,940 candidates, compared to 14,510 in TBB. Here, TBB continues to enjoy a significant performance advantage over TA, NRA and bNRA.

To investigate TBB in more detail, we repeat the experiment with various false positive rates FP. The results are summarized in Table 4. With a looser FP, the memory footprint and the cost of probing the buckets per candidate record are lower; this explains the initial drop in turnaround time from FP=1% to 10%. However, that also weakens the filtering power of the



(a) # Candidates (b) Scan Depth (c) Turnaround Time

Fig. 6 Independent Attributes, Varying k with $q = 2$ and $FP = 10\%$



(a) # Candidates (b) Scan Depth (c) Turnaround Time

Fig. 7 Independent Attributes, Varying q with $k = 10$ and $FP = 10\%$

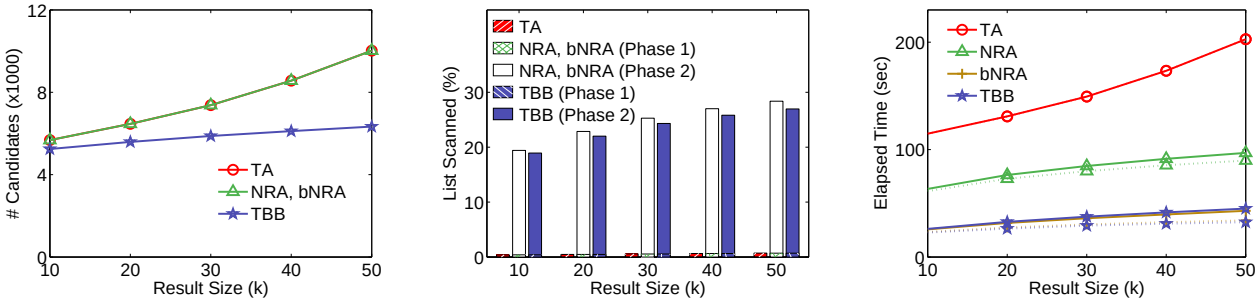
Bloom filters; the consequent higher computation overhead from examining more candidate records in phase two soon cancels out any reduction in per-probe cost, causing the overall processing time to rise. Hence $FP = 10\%$ represents a good compromise.

4.3 Attributes with Positive Linear Correlation

The next experiment is designed to profile the sensitivity of the algorithms to the data distribution. We create a dataset with 10 independent attributes that are generated from a uniform distribution over $[0, 1]$ as before. In addition, 10 positively correlated attributes are introduced. Conceptually, the records are clustered along the line from $[0, 0, \ldots, 0]$ to $[1, 1, \ldots, 1]$ in the hyperspace formed by the correlated attributes. The attribute values are generated with the 'mvnrnd' function in Matlab, using correlation coefficients of 0.5. The dataset contains one million records, while the queries are formulated with randomly selected attributes that are weighted equally. Our TBB algorithm is configured to discount attributes with positive correlations that are significant at 95% confidence level. Figure 8 shows the corresponding results.

Since the records in this dataset tend to have similar values in all the attributes, we might expect the top-scoring records to be confirmed within the first few disk blocks. However, Figure 8(b) clearly indicates otherwise, as NRA, bNRA and TBB all scan much deeper than previously. On closer examination, we discover that with positive correlation, many candidate records are similar enough to each other that their precise aggregate scores are needed to order them. In other words, their upper-bound scores are no longer sufficient. This is why TBB manages to filter out far fewer candidate records in Figure 8(a) than in Figure 6(a). This is why TBB's timings in Figure 8(c) are about the same as those of bNRA. Another observation in Figure 8(b) is that our technique of discounting positively correlated query attributes is effective in enabling TBB to avoid overestimating $depth_{result}$. Overall, TBB is 4.4 to 4.5 times speedier than TA, and between 2.1 to 2.4 times faster than NRA.

For this experiment, we have intentionally picked a high correlation coefficient of 0.5 to create a workload for which it is important for our scheme to avoid overestimating the bucket size b and $depth_{thres}$. We also investigated the sensitivity of TBB to varying correlation levels. We observed that, as the correlation coefficient is lowered, our scheme sometimes over-discounts the correlated attributes. As a consequence, the query processor needs to activate lines 36–37 in Algorithm 3



(a) # Candidates (b) Scan Depth (c) Turnaround Time

Fig. 8 Positively Correlated Attributes, Varying k with $q = 2$ and $FP = 10\%$

to fetch a small number of blocks beyond the estimated $depth_{thres}$ so as to complete phase one. The performance deterioration is marginal, though. If we continue to lower the correlation level, the workload and performance results eventually converge with those for independent attributes in Section 4.2. Overall, TBB still preserves its performance advantage over TA and NRA. Since the results do not provide additional insights, we have omitted them from the paper.

4.4 Attributes with Negative Linear Correlation

Next, we investigate the effect of negative correlation among the query attributes. Each record in this dataset is formulated in two steps: (a) A value $\varrho \in [0, 1]$ is picked from a Normal distribution centered at 0.5. (b) In the hyperspace formed by the attributes, the record is a randomly selected point in the hyperplane that is orthogonal to the line from $[0, 0, \ldots, 0]$ to $[1, 1, \ldots, 1]$ and intersects it at $[\varrho, \varrho, \ldots, \varrho]$. We further impose a constraint that the record must fall within the range of $[0, 1]$ along each dimension. The experiment results are plotted in Figure 9.

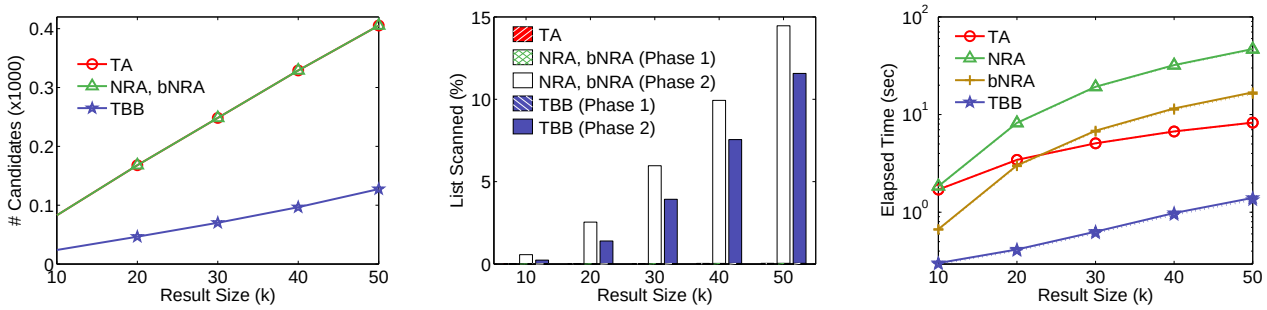
With the Normal distribution, the attribute values in each sorted list fall rapidly in the initial blocks, and most of the attribute values are clustered around 0.5. This characteristic allows the threshold τ to drop below S_k quickly, which explains the small number of candidates as well as low $depth_{thres}$ in Figures 9(a) and Figure 9(b). The situation leads to the negligible CPU costs for all four schemes in Figure 9(c), but TA benefits the most because its I/O cost is also linear to the candidate count. At the same time, the negative correlation among query attributes causes NRA to scan significantly deeper in phase two in order to resolve the candidate records. The two factors combined to make NRA the worst performer in this experiment. Even bNRA, which benefits from sequential I/Os, underperforms TA for $k > 25$. In contrast, the number of candidates in TBB is only roughly 2.5 times the value of k, and Formula 6 enables TBB to disqualify or order all of the candidates without scanning much farther. As a result, TBB continues to lead the other algorithms; its speed-up ranges from 5.6 to 5.9 times with respect to TA, between 6 and 33 times against NRA, and between 2.2 and 12 times versus bNRA.

4.5 Text Corpus

Our fourth experiment uses the WSJ corpus, which comprises 172,961 articles published in the Wall Street Journal from December 1986 to March 1992. After removing stopwords (common words like 'the' and 'a' that are not useful for differentiating between documents) and those that appear in only one article, we are left with 181,978 search terms. (The removal of stopwords and single-document words is a standard procedure in document retrieval [4], and is not a specific requirement of our scheme.) For each term A_j, we create a sorted list L_j of entries $\langle r.A_j, r\rangle$ where $r.A_j$ is the term frequency tf of A_j in document r, multiplied by the inverse document frequency idf of A_j. Similarly, the weight w_j of each sorted list L_j is set to the tf of A_j in the query Q multiplied with the idf of A_j. With this formulation, our top-k query model is equivalent to the classical vector space model [4] for similarity-based text retrieval.

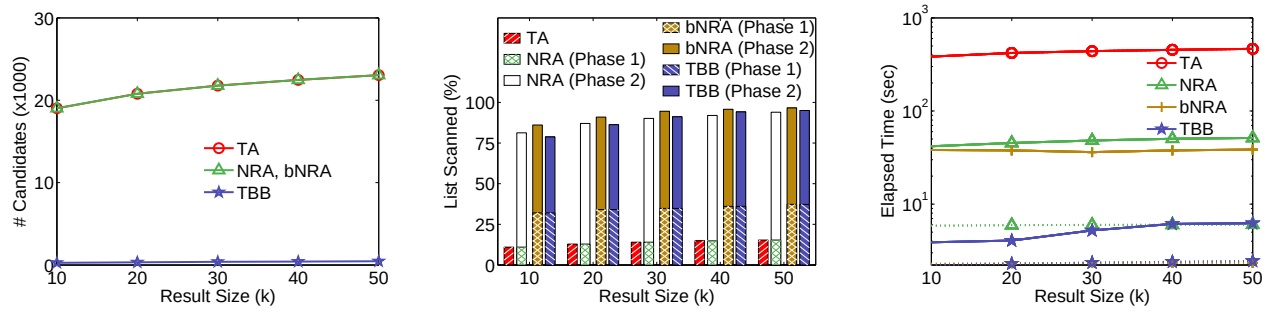
The workload for the experiment is made up of TREC-2 and TREC-3 ad-hoc queries (topics 101 to 200) [40]. The TREC queries contain between two and 20 terms each, and provide realistic term compositions for testing our proposed scheme. Figure 10 summarizes the performance results for $k = 10$.

A property of the WSJ corpus that differentiates it from our earlier datasets is that the sorted lists no longer follow a uniform data distribution. Moreover, the sorted lists are not uniform in length. Figure 11 plots the cumulative frequency of sorted lists (on the y-axis) that are up to various list lengths (on the x-axis).



(a) # Candidates (b) Scan Depth (c) Turnaround Time

Fig. 9 Negatively Correlated Attributes, Varying k with $q = 2$ and $FP = 10\%$



(a) # Candidates (b) Scan Depth (c) Turnaround Time

Fig. 10 TREC Queries, Varying k with $FP = 10\%$

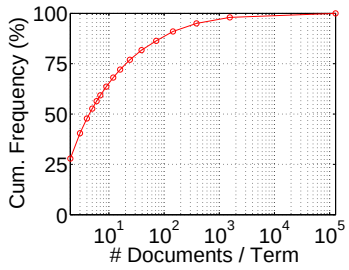


Fig. 11 List Length Distribution of WSJ Corpus

More than half of the lists contain only between two and five entries each, whereas the lists corresponding to the more common words are much longer; in fact, the longest list has 127,848 entries. It is therefore beneficial most of the time to fetch all the constituent blocks of the query lists at once, except for the minority of extraordinarily long lists. The prevalence of short lists is why Figure 10(b) shows that NRA, bNRA and TBB fetch a high percentage of the query lists. As in previous experiments, here the processing time (ranging from 384 seconds to 465 seconds per query) of TA is almost entirely accounted for by I/O time, whereas that of NRA (measuring between 42 and 51 seconds) comprises about 13% I/Os and 87% computation. Again, TBB is able to reduce the I/O cost through sequential disk reads, and the computation cost by filtering out non-viable candidates, thereby achieving overall turnaround times of less than six seconds; this represents a speed-up of 75 to 100 times, 8 to 11 times, and 6 to 10 times relative to TA, NRA and bNRA, respectively.

We have also conducted another document retrieval experiment like the one reported in this section, with a different dataset – the Reuters Corpus Volume 1 [23] specifically. That experiment surfaced consistent performance results and algorithm behaviors as those we observed here for the WSJ corpus.

4.6 Nested Top-k Operations

Our last set of experiments is designed to study queries that involve nested top-k operations, as discussed in Section 3.7. Using the dataset with independent attributes from Section 4.2, we execute the query plan in Figure 5 with the four competing algorithms. The results are summarized in Figure 12.

As shown in the figure, for NRA, bNRA and TBB the I/O cost is dwarfed by the CPU overhead, the latter accounting for the gap between the dotted line (representing the I/O cost) and the solid line (representing the total cost) for the respective algorithms. This is because the input depth ($depth_{result}$) of the second top-k opera-

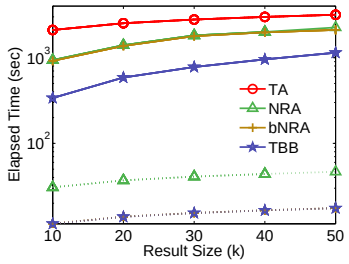


Fig. 12 Nested Top-k Operations

tion op_2 corresponds to the result size of the first top-k operation op_1, causing the latter's input lists A_1 and A_2 to be scanned deeply. For example, for TBB with result size $k = 10$, op_2 pulls in almost 4,400 entries from the output of op_1; this leads to around 93,500 entries being scanned from each of A_1 and A_2, and 109,350 candidate records to be processed in phase two of the algorithm. As a result of the high CPU overheads, the gains enjoyed by TBB is smaller here. Notwithstanding that, TBB's ability to prune non-viable candidates continue to be advantageous, enabling it to be 1.8 times, 2 times, and 2.8 times as fast as bNRA, NRA and RA, respectively.

4.7 Discussion on Experiment Results

The key observations that we gleam from the above experiments are as follows:

- There is no consistent winner between TA and NRA. NRA can be an order of magnitude faster than TA for certain data distributions (seen in Sections 4.2, 4.3 and 4.5), and yet be more than 5 times slower than TA for other distributions (in Section 4.4). Without prior knowledge of the dataset and query workload, it is not possible to choose between the two algorithms.
- Formulas 2, 3 and 6, coupled with our technique for discounting positively correlated query attributes, produce $depth_{threshold}$ and $depth_{result}$ estimates that are robust enough for datasets with widely varied distributions. These estimates enable TBB to reduce I/O cost through sequentially fetching the required portions of the query lists.
- Our bucket organization and the associated Bloom filters are very effective in cutting down the number of candidate records, and in turn the CPU cost. The experiments show that the setting of $FP = 10\%$ strikes a good balance between the per-probe cost and the number of false positives. According to Formula 4, 4.8 bits per sorted list entry are needed to achieve $FP = 10\%$, so the Bloom filters for a dataset with one million records and 20 attributes necessitate just over 500 Kbytes of memory space.
- Overall, our proposed TBB algorithm consistently outperforms TA and NRA for all the datasets that we tested. The speed-up achieved by TBB ranges from two times, to two orders of magnitude. This compares favorably with improvements reported for existing schemes in the literature; for example, [5] showed gains of 1.5 to 3 times in most cases, and up to 5 times at the maximum.

5 Conclusion

In this paper, we address the problem of processing *exact* top-k queries over sorted lists. Such a top-k query generally executes in two phases – find a cut-off threshold for the top-k result scores, then evaluate all the records that could score above the threshold. We introduce a model for estimating the depths to which each sorted list needs to be processed in the two phases, so that (most of) the required records can be fetched efficiently through sequential or batched I/Os. We also devise a mechanism to quickly rank the data that qualify for the query answer and to eliminate those that do not, in order to reduce the computation demand of the query processor. Extensive experiments with diverse datasets confirm that our techniques lead to significant performance gains over existing threshold algorithms, at the expense of a modest memory overhead.

For applications that perform top-k operations directly, it would be straightforward to incorporate our scheme. In other systems, top-k processing may constitute only a step in fulfilling the user request, and it would be interesting to extend our scheme to support the overall query plan. Taking text search engines as an example, we will need to accommodate more elaborate scoring functions such as Okapi [49] and PageRank [7]. There are also many interesting engineering challenges in implementing the scheme as a query operator to support ranked database queries [20].

Acknowledgements HweeHwa Pang is supported by Research Grant 08-C220-SMU-03 from the Singapore Management University.

PROOF(informal) Consider a record r satisfying $r.A_j < \beta_j$. We show that its score is less than S_k. To prove it, we introduce a virtual record r' s.t. $r'.A_j = \theta_j$. To prove the theorem, it suffice to prove that $S(r|Q) < S(r'|Q) \leq S_k$.

References

1. Adomavicius, G., Tuzhilin, A.: Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. IEEE TKDE **17**(6), 734–749 (2005)
2. Akbarinia, R., Pacitti, E., Valduriez, P.: Best Position Algorithms for Top-k Queries. In: VLDB, pp. 495–506 (2007)
3. Arai, B., Das, G., Gunopulos, D., Koudas, N.: Anytime Measures for Top-k Algorithms on Exact and Fuzzy Data Sets. VLDB J **18**(2), 407–427 (2009)
4. Baeza-Yates, R., Neto, B.R.: Modern Information Retrieval. Addison Wesley (1999)
5. Bast, H., Majumdar, D., Schenkel, R., Theobald, M., Weikum, G.: IO-Top-k: Index-access Optimized Top-k Query Processing. In: VLDB, pp. 475–486 (2006)
6. Bloom, B.: Space/Time Trade-Offs in Hash Coding with Allowable Errors. Communications of the ACM **13**(7), 422–426 (1970)
7. Brin, S., Page, L.: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Computer Networks and ISDN Systems **30**(1–7), 107–117 (1998)
8. Bruno, N., Wang, H.W.: The Threshold Algorithm: From Middleware Systems to the Relational Engine. IEEE TKDE **19**(4), 523–537 (2007)
9. Chang, K.C.C., Hwang, S.: Minimal Probing: Supporting Expensive Predicates for Top-k Queries. In: SIGMOD, pp. 346–357 (2002)
10. Chaudhuri, S., Das, G., Hristidis, V., Weikum, G.: Probabilistic Information Retrieval Approach for Ranking of Database Query Results. ACM TODS **31**(3), 1134–1168 (2006)
11. Chaudhuri, S., Gravano, L., Marian, A.: Optimizing Top-k Selection Queries over Multimedia Repositories. IEEE TKDE **16**(8), 992–1009 (2004)
12. Deshpande, P.M., P, D., Kummamuru, K.: Efficient Online Top-k Retrieval with Arbitrary Similarity Measures. In: EDBT, pp. 356–367 (2008)
13. Fagin, R., Lotem, A., Naor, M.: Optimal Aggregation Algorithms for Middleware. In: PODS, pp. 102–113 (2001)
14. Fagin, R., Lotem, A., Naor, M.: Optimal Aggregation Algorithms for Middleware. JCSS **66**(4), 614–656 (2003)
15. Güntzer, U., Balke, W.T., Kiessling, W.: Optimizing Multi-Feature Queries for Image Databases. In: VLDB, pp. 419–428 (2000)
16. Hua, M., Pei, J., Fu, A.W.C., Lin, X., Leung, H.F.: Efficiently Answering Top-k Typicality Queries on Large Databases. In: VLDB, pp. 890–901 (2007)
17. Hua, M., Pei, J., Fu, A.W.C., Lin, X., Leung, H.F.: Top-k Typicality Queries and Efficient Query Answering Methods on Large Databases. VLDB J **18**(3), 809–835 (2009)
18. Hung, H.P., Chuang, K.T., Chen, M.S.: Efficient Process of Top-k Range-Sum Queries over Multiple Streams with Minimized Global Error. IEEE TKDE **19**(10), 1404–1419 (2007)
19. Hwang, S., Chang, K.C.C.: Optimizing Top-k Queries for Middleware Access: A Unified Cost-Based Approach. ACM TODS **32**(1), 5 (2007)
20. Ilyas, I.F., Aref, W.G., Elmagarmid, A.K.: Joining Ranked Inputs in Practice. In: VLDB, pp. 950–961 (2002)
21. Jin, C., Yi, K., Chen, L., Yu, J.X., Lin, X.: Sliding-Window Top-k Queries on Uncertain Streams. VLDB pp. 301–312 (2008)
22. Korn, F., Pagel, B.U., Faloutsos, C.: On the 'Dimensionality Curse' and the 'Self-Similarity Blessing'. IEEE TKDE **13**(1), 96–111 (2001)
23. Lewis, D.D., Yang, Y., Rose, T.G., Li, F.: RCV1: A New Benchmark Collection for Text Categorization Research. Journal of Machine Learning Research **5**, 361–397 (2004)
24. Lian, X., Chen, L.: Top-k Dominating Queries in Uncertain Databases. In: EDBT, pp. 660–671 (2009)
25. Mamoulis, N., Cheng, K.H., Yiu, M.L., Cheung, D.W.: Efficient Aggregation of Ranked Inputs. IEEE ICDE p. 72 (2006)
26. Mamoulis, N., Yiu, M.L., Cheng, K.H., Cheung, D.W.: Efficient Top-k Aggregation of Ranked Inputs. ACM TODS **32**(3), 19 (2007)
27. Marian, A., Bruno, N., Gravano, L.: Evaluating Top-k Queries over Web-Accessible Databases. ACM TODS **29**(2), 319–362 (2004)
28. Michel, S., Triantafillou, P., Weikum, G.: KLEE: A Framework for Distributed Top-k Query Algorithms. In: VLDB, pp. 637–648 (2005)
29. Mouratidis, K., Bakiras, S., Papadias, D.: Continuous Monitoring of Top-k Queries over Sliding Windows. In: SIGMOD, pp. 635–646 (2006)
30. Ntoulas, A., Cho, J.: Pruning Policies for Two-Tiered Inverted Index with Correctness Guarantee. In: SIGIR, pp. 191–198 (2007)
31. Qi, Y., Candan, K.S., Sapino, M.L.: Sum-Max Monotonic Ranked Joins for Evaluating Top-k Twig Queries on Weighted Data Graphs. In: VLDB, pp. 507–518 (2007)
32. Schnaitter, K., Spiegel, J., Polyzotis, N.: Depth Estimation for Ranking Query Optimization. In: VLDB, pp. 902–913 (2007)
33. Shmueli-Scheuer, M., Li, C., Mass, Y., Roitman, H., Schenkel, R., Weikum, G.: Best-Effort Top-k Query Processing Under Budgetary Constraints. ICDE pp. 928–939 (2009)
34. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 7th Edition. John Wiley & Sons (2006)
35. Soliman, M.A., Ilyas, I.F., Chang, K.C.C.: Probabilistic Top-k and Ranking-Aggregate Queries. ACM TODS **33**(3), 1–54 (2008)
36. Spink, A., Wolfram, D., Jansen, M.B.J., Saracevic, T.: Searching the Web: The Public and Their Queries. Journal of the American Society for Information Science and Technology **52**(3), 226–234 (2001)
37. Tao, Y., Xiao, X., Pei, J.: Efficient Skyline and Top-k Retrieval in Subspaces. IEEE TKDE **19**(8), 1072–1088 (2007)
38. Theobald, M., Bast, H., Majumdar, D., Schenkel, R., Weikum, G.: TopX: Efficient and Versatile Top-k Query Processing for Semistructured Data. VLDB J **17**(1), 81–115 (2008)
39. Theobald, M., Weikum, G., Schenkel, R.: Top-k Query Evaluation with Probabilistic Guarantees. In: VLDB, pp. 648–659 (2004)
40. TREC: Text REtrieval Conference. Http://trec.nist.gov/
41. Vlachou, A., Doulkeridis, C., Norvåg, K., Vazirgiannis, M.: On Efficient Top-k Query Processing in Highly Distributed Environments. In: ACM SIGMOD, pp. 753–764 (2008)
42. Xiao, C., Wang, W., Lin, X., Shang, H.: Top-k Set Similarity Joins. In: ICDE, pp. 916–927 (2009)
43. Xin, D., Han, J., Chang, K.C.C.: Progressive and Selective Merge: Computing Top-k with Ad-hoc Ranking Functions. In: ACM SIGMOD, pp. 103–114 (2007)
44. Yi, K., Li, F., Kollios, G., Srivastava, D.: Efficient Processing of Top-k Queries in Uncertain Databases with x-Relations. IEEE TKDE **20**(12), 1669–1682 (2008)
45. Yiu, M.L., Mamoulis, N.: Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data. In: VLDB, pp. 483–494 (2007)
46. Yiu, M.L., Mamoulis, N.: Multi-Dimensional Top-k Dominating Queries. VLDB J **18**(3), 695–718 (2009)
47. Yiu, M.L., Mamoulis, N., Vaitis, M.: Top-k Spatial Preference Queries. In: ICDE, pp. 1076–1085 (2007)

48. Zhu, L., Rao, A., Zhang, A.: Theory of Keyblock-Based Image Retrieval. ACM Transactions on Information Systems **20**(2) (2002)
49. Zobel, J., Moffat, A.: Inverted Files for Text Search Engine. ACM Computing Surveys **38**(2) (2006)
50. Zou, L., Chen, L.: Dominant Graph: An Efficient Indexing Structure to Answer Top-k Queries. In: ICDE, pp. 536–545 (2008)