Best-Effort Top-k Query Processing Under Budgetary Constraints

Michal Shmueli-Scheuer [1,2], Chen Li [2], Yosi Mass [1], Haggai Roitman [1], Ralf Schenkel [3], Gerhard Weikum [3]
[1]IBM Haifa Research Lab, Haifa 31905, Israel
{shmueli,yosimass,haggai}@il.ibm.com
[2]University of California, Irvine, CA 92697, USA
{mshmueli,chenli}@uci.edu
[3]Max-Planck-Institut für Informatik, Saarbrücken, Germany
{schenkel,weikum}@mpi-inf.mpg.de

***Abstract*— We consider a novel problem of top-k query processing under budget constraints. We provide both a framework and a set of algorithms to address this problem. Existing algorithms for top-k processing are budget-oblivious, i.e., they do not take budget constraints into account when making scheduling decisions, but focus on the performance to compute the final top-k results. Under budget constraints, these algorithms therefore often return results that are a lot worse than the results that can be achieved with a clever, budget-aware scheduling algorithm. This paper introduces novel algorithms for budget-aware top-k processing that produce results that have a significantly higher quality than those of state-of-the-art budget-oblivious solutions.**

I. Introduction

A. Motivation

Top-k queries are broadly used in many application areas ranging from queries combining text, metadata, and multimedia; queries that merge streams from sensor readings data; queries combining network statistics; and many more. Consequentially , the top-k problem has been widely studied. The most famous algorithm proposed for efficient computing of top-k results is the TA (Threshold) algorithm proposed by Fagin et al. [9]. Finding the *true* top-k result can sometimes be quite resource-intensive and time-consuming. The main factor in measuring top-k performance is the cost for accessing the lists from the different sources. It is usually assumed that the objects on each source are sorted according to some local score and that it is possible to access the sources either sequentially (sorted access) or by random access. Furthermore, sorted access and random access can vary in cost. For example, for disk-managed lists the cost for random access is much more expensive than sorted access. As another example, in a distributed setting, accessing the lists may require a costly network communication, serving as the main cost factor. Therefore, in such setting, sorted access and random access costs may be quite similar. To minimize such costs, approximate results may be returned instead of the exact top-k results.

In this work we consider a novel problem of top-k query processing under budget constraints. In the presence of a limited budget for top-k query processing, a solution is required to carefully decide on the schedule of data-access tasks. Unwise decisions may in turn result in sub-optimal utilization of the limited budget. As an example, let us assume a top-k query submitted over a collection of images looking for the best k objects combining metadata and image scores. Using sorted accesses only, the system may need to scan large portions of the lists to find the top-k objects that satisfy the query, and breach the budget constraints. On the other hand, random accesses can be quite expensive and should therefore, be used carefully.

We identify top-k queries under budget constraints to be useful for the following emerging applications:

- *Mobile applications*: Mobile applications have highly impatient users, while queries become more and more complex. Consider a user who expects to pass a shopping mall in a few seconds and wants to know if there are any coffee shops near that mall. Once she passes the mall, the results are no longer useful. In this case, the availability of the user near the mall constraints the time on which the system can answer the user query. To be capable to satisfy the user's query on time, the system may return approximate results rather than the exact results which computation may bypass the time limit.
- *Real-time analytics*: IP-traffic logs in network monitoring or click-stream logs in search and web-business generate huge volumes of data. Their analysis may be driven by trial-and-error. Thus, for example, an analyst that wishes to analyze the data and take decisions accordingly, may require fast approximate answers. The queries would typically be generated by some visual-analytics tools. Many such queries could be triggered in a short time and the user would expect all of them to be answered in an interactive manner - so that the visualization refreshes in a second or two.

In the rest of this paper we address the problem top-k query processing under budget constraints and provide both a framework and a set of algorithms to address this problem.

B. Problem Statement and Contributions

The above scenarios emphasize the need for new and efficient top-k query processing methods when working with a limited budget, in terms of time or access costs. Existing work on top-k processing has focused on achieving exact results or at least approximate results with a certain quality with minimal costs. In contrast, this paper focuses on the dual problem: instead of trying to minimize the processing costs to reach a target result quality, we provide a best-effort top-k query processing given a fixed limit for the execution cost. Such a budget can be easily defined in terms of time, number of disk accesses, or number of network messages, reflecting an essential limitation faced by contemporary applications. We show that prior knowledge of the total available budget can be used to achieve a better top-k precision than a budget-oblivious top-k algorithm that does not have such knowledge in advance.

This work makes the following contributions:

- We present a novel framework for approximate top-k query processing under budgetary constraints.
- We develop a set of budget-aware algorithms that use only sorted accesses.
- We present an adaptive budget-aware algorithm for interleaving sorted and random accesses.
- We provide efficient algorithms to calculate the exact solution of the offline problem and use it to show the high quality of our algorithms.
- We conduct extensive experiments using real-world data from TREC and IMDB to evaluate and verify our algorithms.

The rest of this paper is organized as follows. Section II discusses related work. Section III formally defines the problem. In Section IV we provide algorithmic solutions where only sorted accesses are allowed. In Section V, we study the case where random accesses are also permitted. Section VI provides efficient algorithms to compute the optimal results that can be achieved within a given budget. In Section VII, we experiment with real data sets and show the effectiveness of the proposed methods. Finally, we discuss future work and conclude in Section VIII.

II. Related work

Efficiently processing top-k queries has been a very active research area recently. Ilyas et al. [14] give a comprehensive survey of the work in the area. The majority of the proposed algorithms is based on the family of threshold algorithms (TA) [9], [12], [16], extending and improving it in various respects [5], [6], [7], [8], [15]. These algorithms aim at computing the exact top-k results with minimum cost under different settings and assumptions. More specifically, the Upper [6], MPro [7], and Pick [15] algorithms suggest random-access scheduling for cases where sorted accesses are limited or impossible. Marian et al. [15] further show how to consider different access costs, both random and sorted. Bast et al. [5] suggest a solution that minimizes the cost given that all the index lists support both sorted and random accesses and that different lists share the same cost. All these studies essentially address the *dual problem* of our problem: they minimize the cost needed to provide the exact top-k results. Our work, on the other hand, tries to maximize the precision given a budgetary limitation.

A second line of proposals aims at improving the performance of top-k query processing by returning approximate instead of exact top-k results. Probably the first such algorithm was proposed by Theobald et al. [17], which also provided probabilistic guarantees for the quality of the results. This work was empirically shown to consume less budget than exact top-k algorithms, but it did not provide upper bounds for the processing costs. Aray et al. [3] provide anytime top-k query measures for answering top-k queries with probabilistic guarantees. Their algorithm collects statistics during processing, which can be used to provide probabilistic guarantees at any time during processing. For the threshold algorithm (TA), [3] achieves quite similar guarantees as the probabilistic top-k method of [17]. Our work differs from [3] in several important aspects. First, our computational model is different. The method from [3] does not have any prior knowledge on its stopping point; thus, it needs to prepare for being stopped immediately at any time. However, in our setting, we know the deadline and our main focus is to optimize the result quality given that deadline (and only for this deadline, not for any possible stopping time). Second, [3] uses the TA algorithm as is and only adds efficient methods to collect enough statistics for estimating the results approximation, where no additional effort is invested to improve the algorithm execution or minimize the costs. In our work, we propose a set of new algorithms for scheduling sorted and random accesses.

Güntzer et al. [13] developed heuristic strategies for scheduling sorted accesses over multiple lists. These are greedy heuristics based on estimates of scores, which tend to prefer index lists with a sharp gradient. The goal in our work is different, leading to different strategies. Al-Hamdani et al. [1] considered topic-selection queries on top of meta-data. Their output is a list of k answers such that each answer is above a pre-defined threshold. Their approach tries to maximize the number of highest importance-scored output tuples and maximize the size of the query output.

The work in [4] shares some similarity with our work. The authors show how to perform top-k queries for mobile devices, by using a cost-adaptive algorithm (SR) that optimizes object accesses and query run time. Their assumptions, however, are different, since they report on the exact k best results and allow immediate output of objects to the user.

III. Model and Problem Definition

In this section, we present our model, provide the required background and the terminology that is used throughout the paper, and then formally define our problem.

A. Data Model and Notation

We assume a collection of n objects, each with at most m attributes such that each attribute is managed by a source. For example, attributes might be textual terms, metadata, color, texture, etc. Given a query that specifies conditions on at most m attributes, it is possible to get from each source a list of objects sorted by their relevance to the appropriate queried attribute. For the remainder of this paper, the terms *source* and *list* are used interchangeably. Therefore, we assume a set of m lists $L_1, L_2, \ldots, L_m$, one per attribute. Each list L_i consists of pairs of object id and its relevance score $(oid, s_i(oid))$, and is sorted in a non-increasing order of $s_i(oid)$. We assume that object scores on each list are normalized such that they are in the range of $[0, 1]$, where a score of 1 implies the highest relevancy and 0 is the lowest. Given a query and some monotonic aggregation function defined over the set scores of the query attributes, we want to retrieve the top-k most relevant objects. For ease of presentation, we will consider only summation as the aggregation function in the rest of this paper.

B. Algorithmic Framework

A list can be accessed with *sequential* (also known as *sorted*) or *random* accesses. In a sorted access, the next object in descending order of score is retrieved from the associated list. A random access retrieves the score of a given object from the list. We assume that the cost of accessing via a sorted access is C_s, while for a random access the cost is C_r. Such costs can be defined in terms of time (*e.g.,* response time) or disk access (I/O). In a typical centralized application with very long lists that are kept on disk, C_r is often orders of magnitudes larger than C_s [5]. This fact may be different in distributed applications where roundtrip times dominate the access cost, but even there sorted accesses are usually cheaper because they can be executed in batches, amortizing the roundtrip time over many accesses. For ease of presentation, we assume that the costs are independent of the list; all algorithms presented later can be easily adapted if the costs differ between lists. Additionally, we assume that the costs are normalized such that $C_s = 1$.

We consider algorithms within the framework of Threshold algorithms [9], [12], [16]. Here, an algorithm performs a mixture of sorted and random accesses to the lists. At any time during the execution of such an algorithm, some objects may have been only partially seen in a subset of lists, so there is some uncertainty about the final score of the object. The algorithm therefore keeps, for each seen object o, two values to bound its final score: worst score *worstScore(o)* and best score *bestScore(o)*. Here, *worstScore(o)* is computed as the sum of the seen scores of *o*, assuming a score of 0 for the remaining dimensions, and serves as a lower bound for o's final score. *bestScore(o),* is computed as the sum of *worstScore(o)* and the $high_i$ values of lists where o has not yet been seen. $high_i$ is the value at the current scan position of list i. *bestScore(o)* is therefore an upper bound for o's final score. Objects are then kept in two sets: The k objects with the currently highest worst scores form the current top-k answers, and the remaining objects reside in the candidate set. We assign to $mink$ the smallest worst score of any item in the current top-k answers, and any candidate c can be pruned as soon as $bestScore(c) \leq mink$, because it cannot move into the final top-k answers. The algorithm can safely stop if the queue is empty and no yet unseen object can move into the top-k answers, i.e., $\sum high_i \leq mink$.

Instances of algorithms within this framework differ in the operations they are allowed to use, in the scheduling of sorted accesses, and in the scheduling of random accesses. The TA algorithm initially proposed by Fagin et al. [9], for example, uses round-robin scheduling for the sorted accesses. As soon as an object is seen for the first time on any list, TA performs random accesses to the remaining lists in which the object has not been seen yet to evaluate the complete score of that object. As TA always completely evaluates the score of an object, score bounds are always exact, so there is no need to maintain an explicit candidate set in the algorithm. NRA is a version of TA with sorted accesses only, and CA performs a balanced number of sorted and random accesses [10]. The more recent KBA algorithm uses a knapsack-based scheduling for sorted accesses [5].

The *execution trace* (or *trace* for short) for a query of an algorithm within this framework consists of the sequence of steps performed by the algorithm, where each step can be either a sorted access or a random access to a list. Following earlier literature, we assume that random accesses occur only to objects that have been seen by at least one sorted access before (called "no wild guesses" in [10]). We assign to each step its corresponding cost (C_s or C_r). The cost of the whole trace is then the sum of the cost of each step. The length $|t|$ of a trace t is the number of operations it contains, and the results R_t of a trace are the top-k results after the steps of the trace have been executed.

The scope of this paper are *budget-keeping* algorithms, i.e., algorithms that produce only traces whose cost is not above a given budget B. Any top-k algorithm can be made budget-keeping by stopping its execution as soon as the budget would be exceeded. However, these algorithms are at the same time *budget-oblivious,* as they do not take the budget into account when scheduling their operations. In contrast, our new algorithms are *budget-aware,* i.e., they consider the available budget when making scheduling decisions.

A small budget is usually not enough to compute the exact top-k results of a query. Budget-keeping algorithms are therefore inherently approximate in nature. Our primary measure for the quality of results which such a budget-keeping algorithm produces is the overlap of the set R of results of the algorithm with the set R_{exact} of results of an unconstrained top-k algorithm for the same query, measured as relative precision $\frac{|R_{exact} \cap R|}{|R|}$.

C. Problem Definition

Problem 1 (Best-effort top-k query processing): Given a query q and a budget B, we would like to find an execution trace that (1) has cost at most B, and (2) where the relative precision of the result is maximal among all other traces with cost at most B.

This problem setting is fundamentally different from the traditional, unconstrained, setup for the TA-family methods. If we merely ran TA (or NRA or CA) and stopped it when the budget expired, the results might be significantly inferior to those of a budget-aware method. This is because the standard TA might pursue a strategy in which the clear picture about the final top-k results only emerges towards the very end of the algorithm, often long after the budget is exhausted. For example, one of the best scheduling methods for TA-style algorithms is to postpone random access as much as possible. In similar fashion, the rationale for scheduling sequential accesses changes, too; traditional TA methods know that they run their computation to natural completion anyway and can thus afford to be "patient", whereas our setting requires that we perform the "most insightful" accesses before we run out of budget.

For analogous reasons, the budget-constrained problem - the best possible approximation of result quality after a constant number of steps - is also *not* simply a dual problem of the classical problem with approximation - shortest number of steps for a given approximation ratio of the result quality [10], [17]. Methods for approximate top-k querying do not carry over to our setting.

Finally, although there is a latent connection to the anytime top-k problem addressed in [3], the problems again are fairly different. Anytime algorithms aim to produce a good approximation for any possible point of termination, which may force them to be overly greedy. In our setting, we know when the budget will be exhausted, so we can carefully schedule requests with adequate foresight but limited patience.

In the remainder of this paper, we describe the budget-aware algorithms we developed with traces that are close to the optimal traces. We start with algorithms that perform only sorted accesses (Section IV), followed by algorithms that also consider random accesses (Section V).

D. An Example

Figure 1 illustrates an example with objects from two lists aggregated using summation, requesting the top-2 results. Assume that each sorted access has a cost $C_s = 1$, and each random access has a cost of $C_r = 3$. The exact result is $\{d, t\}$, but depending on the budget, a budget-keeping algorithm cannot always compute this result. For example, if the budget is below 7, no budget-keeping algorithm can return d in the top-2 results. There is simply not enough budget to read d with sorted accesses on both lists. When d is seen on one list, a budget of at least 4 should have been spent, so there is no enough budget to do a random access to d on the other list. The score of d is therefore always below the scores of at least two other objects. On the other hand, we need a budget of at least 6 to return t (three sorted steps in L_1, followed by a random access to t in L_2). An optimal trace has cost 10, which includes 4 sorted steps in L_1 and two random accesses in L_2.

The budget-oblivious algorithms TA and NRA need a higher budget to get a result with a similar quality. TA has t in its top-k set after five sorted accesses (three to L_1, two to L_2) and five random accesses (to s, u, and t in L_2, and to a and b in L_1), requiring a budget of at least 20 for a precision of

	L_1	L_2
1	s:0.95	a:1.00
2	u:0.93	b:0.90
3	t:0.92	c:0.85
4	d:0.90	d:0.80
5	x:0.50	e:0.70
6	y:0.40	t:0.60
7	z:0.20	f:0.40
8	⋮	⋮

Fig. 1. Prefixes of two lists

0.5. Correspondingly, a budget of 28 is required until TA has put d into its top-k list, and hence reached a precision of 1. NRA, which does only sorted accesses in a round-robin order, needs 8 steps to move d into its top-k list, yielding a precision of 0.5, and 12 steps to additionally move t to the top-k for a precision of 1. Note that the unlimited version of NRA would require another 2 scans to eliminate all candidates before it can stop.

It is evident from this example that budget-aware algorithms can be better than budget-oblivious ones. This is beacuse of two factors: clever scheduling of sorted accesses in a non-round-robin manner, and clever scheduling of random accesses.

IV. Sorted Access

We now consider the case where a budget-aware algorithm is allowed to generate traces that contain only sorted accesses. The algorithm needs to decide in each step which list it accesses next, given the current top-k candidates, $high_i$ bounds, and possible statistics of the lists in general. We introduce two different objectives for heuristically selecting a list for the next sorted access.

The first important observation is that an algorithm that prefers accessing lists with high scores has a better chance to find objects that belong to the final top-k results, and hence obtain a higher precision. Intuitively, an object with high scores on the lists where that object has already been seen has a higher chance to be part of the top-k results than an object with "mediocre" scores. Moreover, given the limited budget, an object with mediocre scores that is indeed part of the top-k results would have a low chance to be accessed anyway, since it requires many sorted accesses, possibly more than the budget. Thus, our first objective should be to *prefer high scores*.

After some accesses, as the scores on the lists decrease, we might reach a saturation-like state in which the scores on the lists are not high, but not low either, so they cannot help to differentiate between the candidates. At this point, we define a second objective that *prefers large score reductions*. The idea is that large score reduction means that we can get into an "area" of low scores very quickly. Thus we can stop scanning those lists since low scores can not increase $mink$ nor can they introduce new candidates to the top-k results.

We use a motivating example to illustrate the rational behind this approach. Consider the example depicted in Figure 2.

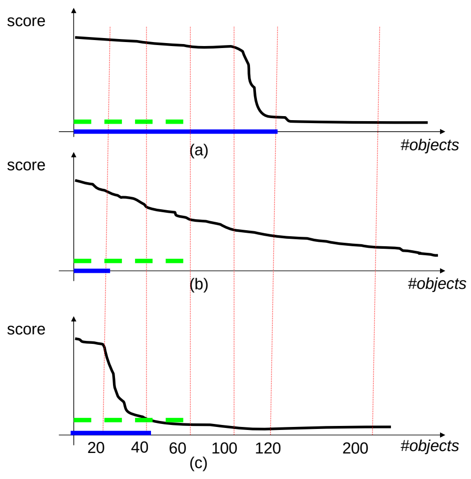


Fig. 2. Three lists with different score distributions

Assume that we have a budget of $B = 180$, i.e., we are allowed to perform up to 180 sorted accesses. Also assume we are given three lists denoted by (a), (b), and (c) in Figure 2, each with a score distribution as depicted in Figure 2. Specifically, list (a) starts with many high-score items, followed by a sharp decrease in their scores (after 100 objects). List (b) has its scores constantly decreasing. Finally, list (c) has a few items with high scores followed by a sharp decrease with many low-score items. The dashed line represents a uniform allocation of 60 sorted accesses for each list; this would be the result of a round-robin scheduling of sorted accesses. The bold line shows a non-uniform allocation of sorted accesses for each list. List (a) has 120 sorted accesses, list (b) has 20 accesses, and list (c) has the remaining 40 sorted accesses. If we perform less than 120 accesses for list (a), then we will miss the sharp decrease, which helps us to prune more unseen and partially seen objects from the set of candidates. However, there is no need to spend more than 120 sorted accesses on list (a), since the added value of these objects will not be large. For list (c), we only spend 40 sorted accesses. Again, after the sharp drop, we mainly encounter low-score items that probably neither help much in adding elements to the top-k results nor in increasing *mink*. Therefore, it would be better to allocate more sorted accesses for list (b) and obtain some more high-score objects.

To summarize, we aim to develop a solution that considers both objectives:

- *Maximize seen object scores*;
- *Maximize score reductions*;

Thus we have a bi-objective optimization problem. In the remainder of this section, we provide two approximate solutions to this problem.

A. Adaptive scheme for bi-objective optimization

Our first solution to the bi-objective optimization problem is based on a conventional optimization technique that combines both objectives into a single weighted objective. This in turn allows us to control the relative importance we assign to each of the original objectives. We develop an adaptive weighting scheme, where the weights can be adjusted during the runtime, which allows us to adaptively change the importance of each objective according to the feedback from current candidate objects in the candidates queue.

We associate two utility functions with each list L_i, corresponding to the two objectives mentioned above. The first one is an *average score utility function* $util_{as}(L_i, x)$, which captures the first objective (maximizing seen objects' scores). Given a budget of x sorted accesses on list L_i, the function returns the average expected score of the x next objects to be seen on L_i. Note that the scores of objects at positions that were not reached yet can be only approximated using expected scores. Expected scores are computed using histograms of the score distributions in the list, similarly to [17]. The average score utility function of performing a batch of x sorted accesses to a list L_i is given by

$$util_{as}(L_i, x) = \frac{1}{x} \cdot \sum_{j=pos_i}^{pos_i+x} score_i(j). \qquad (1)$$

In the formula, x is the budget, pos_i is the current position on list L, and $score_i(j)$ is the expected score of the object located at position j on list L_i. Note that since $score_i(j) \in [0, 1]$ for every j, we have $util_{as}(L_i, x) \in [0, 1]$.

The second function is a *score-reduction utility function* $util_{sr}(L_i, x)$, which captures the second objective (maximizing score reduction). Given a budget of x sorted accesses to be performed on list L_i and the current position pos_i on list L, the function returns the score reduction between the last seen object at position pos_i and the (expected) score at position $pos_i + x$. The score reduction utility for list L_i is given by:

$$util_{sr}(L_i, x) = high_i - score_i(pos_i + x). \qquad (2)$$

Note that $util_{sr}(L_i, x)$ also takes values in $[0, 1]$.

We now combine the two utility functions using a smoothing parameter α, which controls the relative importance we assign to each objective:

$$util(L_i, x) = \alpha \cdot util_{as}(L_i, x) + (1 - \alpha) \cdot util_{sr}(L_i, x). \quad (3)$$

The adaptive scheme works in batches of b sorted accesses at a time. After each batch, we adaptively adjust the weights we assign to each objective using the feedback of the set of the candidate objects that were seen on the different lists. Therefore, the total number of times we adjust the weights during runtime is $\left\lfloor \frac{B}{b} \right\rfloor$.

To allocate a batch of b sorted accesses to the different lists at some iteration $l = 1, 2, \ldots, \left\lfloor \frac{B}{b} \right\rfloor$ of our scheme, we need to find a set of scan depths for the lists in order to maximizes the total gained utility. Formally we solve the following optimization problem:

Problem 2 (Budgeted Sorted Access Scheduling (BSAS)): Given a batch size of b sorted accesses, find an allocation $b_1, \ldots, b_m$ with $\sum_{i=1}^{m} b_i = b$ that maximizes $\sum_{i=1}^{m} util(L_i, b_i)$.

Theorem 1: The BSAS problem is NP-Hard.

This theorem can be proven by reducing the 0-1 knapsack problem to BSAS; the proof can be found in Appendix A. In the following sections we first show how α can be adapted during runtime, and then propose two heuristic scheduling policies that provide approximate solutions to the BSAS problem.

B. Adaptive Calculation of α

According to the two objectives, we identify two major phases during runtime. The first phase is the *gathering phase*, where we aim at finding good candidates for the final results. This phase starts at the beginning of the processing and fades out once we have seen most (or all) potentially good results. In this phase we prefer candidates with high scores. The second phase is the *reduction phase*, where we aim at reducing the set of candidates by reducing their best scores below the $mink$ threshold, which can be done by reducing the $high_i$ values. Therefore, at this phase, we prefer to read from lists where the score reduces quickly. There is no sharp transition between the two phases; instead, the gathering phase slowly evolves into the reduction phase.

To implement this behavior, we adjust the smoothing parameter α in the gathering phase. We give a higher weight to $util_{as}$ in the gathering phase and to $util_{sr}$ in the reduction phase. We set α to 1 when we have not yet seen k different objects (because any new object will immediately move to the top-k results), and to the average probability $\widehat{p_k}$ of the candidate objects c in the candidate set to get into the top-k otherwise, where

$$\widehat{p_k} = \frac{1}{|cand.\ set|} \sum_{c \in cand.\ set} p_k(c) \tag{4}$$

We will show later how to estimate the probability $p_k(c)$ of an object in the candidate set to get into the top-k, based on statistics on the distribution of scores in the lists.

At the beginning of the execution, the chance of the candidates to get into the top-k is relatively high, so $\widehat{p_k}$ is relatively high, too. As the execution progresses $\widehat{p_k}$ decreases.

Figure 3 presents an example of the different values of $\widehat{p_k}$ for a given candidate set with respect to the spent budget at every step during our scheme runtime. At the beginning we still need to gather more candidates c, so more weight (α) is given to the first objective (i.e., maximizing the seen objects' scores). As we proceed, the number of candidates c in the candidates queue increases and $\widehat{p_k}$ decreases as the $high_i$ bounds drop, and therefore, we assign more weight ($1-\alpha$) to the second objective (i.e., large score reductions). For example, in Figure 3, after we perform 1500 SAs, we assign a weight of $\alpha = 0.1$ to the first objective and a weight of $1 - \alpha = 0.9$ to the second objective.

We now describe how we estimate the probability $p_k(c)$ of a given object c in the candidates queue. Given the *worstScore(c)* of c, the current *mink* and the current $high_i$ bounds of the lists, the probability that c will qualify for the top-k can be estimated using the assumed distribution of the remaining unseen scores in the different lists where c has not been seen yet. We denote

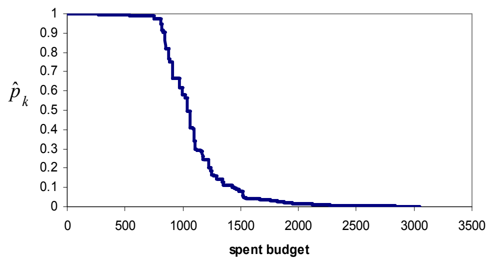


Fig. 3. $\widehat{p_k}$ vs. spent budget for TREC query, $k = 100$

as $E(c)$ the subset of the lists $L_1, L_2, \ldots, L_m$ in which c has already been seen, and as $\bar{E}(c)$ the subset of the lists in which c has not been seen yet. For each missing list $L_i \in \bar{E}(c)$, a random variable S_i represents the score of any remaining object in that list.

Thus, the probability $p_k(c)$ of a candidate object c to get into the top-k is given by:

$$p_k(c) = P\left[\sum_{i \in \bar{E}(c)} S_i > \delta(c) | S_i \leq high_i\right] \tag{5}$$

with $\delta(c) = mink - worstScore(c)$. In our model we use histograms to capture the distributions of the different lists. To derive $p_k(c)$ from these histograms, we use methods similar to those introduced in [17].

C. Fair Heuristic

A first, greedy approximate solution to Problem 2 is the `Fair` heuristic. Given a batch size of b SAs, this heuristic fairly allocates SAs for each list L_i according to the relative estimated utility of that list compared to the other lists, assuming that we could spend the whole budget on each list. Therefore, for each list L_i we calculate the estimated value of $util(L_i, b)$. We then divide the total budget fairly between the m lists according to their relative utilities, so the number of SAs for each list L_i (denoted SA_{L_i}) is:

$$SA_{L_i} = b \times \frac{util(L_i, b)}{\sum_{j=1}^{m} util(L_j, b)} \tag{6}$$

D. Ranking Heuristic

We now present the `Ranking` heuristic that does not consider absolute score values, but relative ranks of objects. In addition, whereas the Fair heuristic does not consider the budget, the `Ranking` heuristic considers the overall budget B when creating the ranked lists.

The `Ranking`-based approach is given in Algorithm 1 and Algorithm 2. The first algorithm, Algorithm 1, generates two ranked lists, given the total budget B. Objects in the *score* list S are ranked by their estimated score, while in the *delta* list D the same objects are ranked by their estimated delta from the object following them in the original lists; ties are broken by listid. We denote the rank in the score list S of an object d by $rank_S(d)$ and the rank in the delta list D by $rank_D(d)$. Note that since we don't actually read the input lists at this time, we

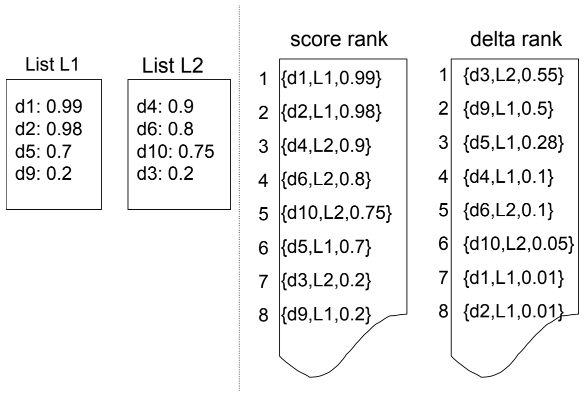


Fig. 4. lists L_1 and L_2, and their corresponding ranked scores and ranked delta lists

can only estimate the scores using the score distributions; the object IDs are just arbitrary IDs used to link the objects in the two generated ranked lists and do not reflect real object IDs. For each of the m input lists $L_1, L_2, ..., L_m$, the algorithm estimates scores for all the not yet seen objects until depth B. For each such object it creates a triplet that consists of the object's id, the list id, and the object's estimated score $\{$oid,Lid,score$\}$ and inserts it into the score list, and another triplet with its estimated delta $\{$oid,Lid,delta$\}$ that is inserted into the delta list. Thus, in each of the two ranked lists, we have a total of $B \times m$ triplets.

Figure 4 illustrates an example with two lists and the corresponding ranked score and delta lists. The creation of the ranked score list is trivial and is performed using the estimated score of each object. The creation of the ranked delta list is performed by calculating the difference between any two consecutive objects. For example, the ranked score list has 8 objects ordered by the score value. The first ranked object is $d1$ from list L_1 with a score of 0.99, followed by $d2$ from list L_1 with a score of 0.98. In the third place, the object is $d4$ from list L_2 with a score of 0.9. The first object in the ranked delta list is $d3$ from list L_2 with delta of 0.55 from the precedence object $d10$ ($0.75-0.2$). The score rank of object $d1$ from list L_1 is $rank_S(d1) = 1$ and its delta rank is $rank_D(d1) = 7$. The score and delta ranks for object $d4$ from list L_2 are $rank_S(d4) = 3$ and $rank_D(d4) = 4$ respectively.

We now explain how we generate the allocation given the ranked lists. Generally speaking, given that we have budget=B, we perform the allocation step by step. The main idea of the allocation algorithm is that on each step, we consider the "best" objects from each list, combining the ranking of the object's delta and score, and choosing the best.

For example, let us assume that $B = 3$, $\alpha = 0.9$, and the lists are as they appear in Figure 4. Doing one step in list L_1 will result in choosing $d1$ with score and delta ranks of 1 and 7, respectively (combined 1.6). Doing one step in list L_2 will result in choosing object $d4$ with a ranking of 3 and 4 (combined 3.1). Thus, we chose to do 1 access in L_1. Now the remaining budget is $B = 2$ and we considered $d2$ from list L_1 and $d4$ from list L_2. Again, the best option is to do one more step in L_1. Finally, $B = 1$ and we compare $d5$ from list L_1 and $d4$ from list L_2 and choose to do one step in list L_2. The final allocation will be 2 accesses on list L_1 and 1 on list L_2. Algorithm 2 formally describes this approach.

Algorithm 1 Generate Rank lists

```
 1: Input: $\{L_1, L_2 \ldots, L_m\}, b$
 2: Output: 2 sorted ranked lists: $S$,$D$
 3: Initialization:
 4: for all lists $L_i$ do
 5:    l=1;
 6:    while $l \leq b$ do
 7:       generate $\{$oid,$L_i$,$score_i(pos_i + l)\}$;
 8:       generate $\{$oid,$L_i$,$score_i(pos_i + l - 1) - score_i(pos_i + l)\}$;
 9:       add $\{$oid,$L_i$,$score_i(pos_i + l)\}$ to list $S$;
10:       add $\{$oid,$L_i$,$score_i(pos_i + l - 1) - score_i(pos_i + l)\}$ to list $D$;
11:       $l \leftarrow (l + 1)$;
12:    end while
13: end for
14: sort $S$
15: sort $D$
```

Algorithm 2 Create Allocation

```
1: Input: $rank_S$, $rank_D$, B,$\alpha$
   Output: $acc_i$, number of accesses for each list $L_i$
2: while $l \leq B$ do
3:    for all lists $L_i$ do
4:       let $d_i$:= object with highest $rank_S(d_i)$ from list $L_i$
5:    end for
6:    $i = argmin_i : \{\alpha * rank_S(d_i) + (1 - \alpha) * rank_D(d_i)\}$
7:    $acc_i \leftarrow (acc_i + 1)$;
8:    $l \leftarrow (l + 1)$;
9: end while
```

V. Random Access

Random accesses are often beneficial in unconstrained top-k algorithms to reduce the overall execution cost or runtime, since they help to detect missing scores of promising candidates that otherwise would require heavy scanning of the lists [10]. In budget-aware algorithms, they can have a similar positive effect on result quality, at least if the budget is high enough to perform a reasonable amount of them (as we will further demonstrate in Section VII). This section introduces a heuristic scheduling strategy for random access that assigns, similarly to the LAST strategy introduced in [5], a certain budget at the end of the execution to random accesses.

We assume for ease of presentation that we can perform random access on any list $L_1, L_2, \ldots, L_m$ (if that was not possible for some lists, we would simply ignore those lists when scheduling random accesses), where each random access has a cost of C_r. We advocate an execution with two phases. In the first phase, we perform only sorted accesses to gather "enough" good candidates for the results, using any of the strategies introduced in Section IV. In the second phase, we perform only random accesses to promising objects in the candidate set to identify the final results. We now explain in detail how to determine the right point to switch from the first to the second phase, and how to select "promising" objects for random access.

A. *Switching from SA to RA*

Inspired by the LAST algorithm [5], we seek to adaptively balance the number of RAs and SAs with respect to the cost ratios. The LAST algorithm spends approximately the same budget on SA and RA, switching from SA to RA when the expected cost for future random accesses does not exceed the aggregated cost for sorted accesses done so far. As a non-approximating algorithm, LAST can perform the SA-RA switch only if it is guaranteed that all potential results have been encountered by random access, i.e., when $\sum_{i=1}^{m} high_i < mink$; This corresponds to the time when the gathering phase (see Section IV-A) finishes. However, in the presence of a limited budget, this condition will typically not be true before we run out of budget, and we will even more rarely reach the point when LAST would switch from SA to RA. We therefore propose an adaptive budget for random accesses, capturing the tradeoff between gathering enough good candidates with sorted accesses and keeping enough budget for the random accesses. Remember that at the end of the gathering phase, we would be willing to spend as much cost for RA as we did for SA before, and that α introduced in Section IV-A quantifies how much of the gathering phase we have already finished. Our heuristics now assigns more budget to random access if we have proceeded further into the gathering phase. Formally,

$$R = (1 - \alpha) \cdot S \tag{7}$$

where S is the total sorted-access cost so far. Note that, as a consequence of this definition, we will not spend any budget to random accesses unless we have seen at least k different objects (because $\alpha = 1$ then). We switch from the sorted-access to the random-access phase as soon as the total sorted-access cost S so far plus the budget R assigned for random accesses exceeds the available overall budget, i.e., $S + R > B$.

B. *Scheduling Random Accesses*

After switching from sorted to random accesses, the algorithm has a list of candidates that were partially seen in the SA phase but could not make it into the top-k since their *worstScore* is less than *mink*. We are left with a random-access budget to perform at most $\lfloor R/C_r \rfloor$ random accesses, and we need to select the most promising candidates upon whom to perform those accesses. This is done as follows: for each object c in the candidate set, we calculate its expected score

$$expScore(c) = worstScore(c) + \sum_{l \in \bar{E}(c)} (\hat{S}_l(c))$$

where $\hat{S}_l(c)$ is the expected score of c in an unseen list L_l, calculated using the histograms. We then choose the object c with the maximum expected score and perform a random access on the missing list L_l where $\hat{S}_l(c)$ is maximal. Then score bounds of c, the top-k list and the set of candidates are updated accordingly, and the algorithm continues selecting the next candidate object for random access, until the budget has been exhausted.

VI. Efficiently Solving the Offline Optimization Problem

This section presents efficient algorithms to solve the offline version of Problem 1, i.e., given a query q, its results R_{exact} with unlimited budget and a budget B, find a trace with cost at most B whose precision is optimal among all other such traces. We will use the precision of this trace later in the experiments to compare the results of our algorithms with the best possible precision. We first present in Subsection VI-A a solution for sorted-access traces and then extend it to traces with sorted and random accesses.

A. *Optimal Solution for Sorted-Access Traces*

Formally, we consider for a query q with lists $L_1, \ldots, L_m$ the set of possible sorted-access traces T, i.e., traces that contain only sorted accesses to $L_1, \ldots, L_m$. For ease of notation we write t_i for the number of sorted accesses to L_i in trace $t \in T$. We assume that we know the top-k result R_{exact} of the query with unlimited budget. Given a budget constraint B, we now aim to find any $t \in T$ with $|t| \le B$ that maximizes relative result precision, i.e.,

$$argmax_{t \in T \wedge |t| \le B} \frac{|R_t \cap R_{exact}|}{|R_t|}$$

where R_t denotes the result (top-k) of trace t. We call that trace *opt* and its result R_{opt}.

It is prohibitively expensive to try all possible traces, but it is enough to consider only a (relatively) small subset of them. Recall that we know the set R_{exact} of results of the unbounded top-k process. For a list L_i, we consider the set P_i of positions in L_i where objects from R_{exact} occur. The following lemma shows that it is enough to consider traces t where, for each list L_i, $t_i \in P_i$:

Lemma 2: Given a budget B, there is at least one sorted-access trace t with $|t| \le B$, $\frac{|R_t \cap R_{exact}|}{|R_t|}$ maximal among all sorted-access traces within the budget, and $t_i \in P_i$ for all i.

Proof: Assume that such a t does not exist. Then pick any trace s among the traces within the budget that have optimal relative precision, and consider trace s' where s'_i is set to the largest position from P_i less or equal to s_i. The score of any object in R_{exact} is the same after the execution of s and s' as the two traces differ only in operations on documents not in R_{exact} by construction. Now $R_{exact} \cap R_s \subseteq R_{exact} \cap R_{s'}$ because the score of any object after the execution of s' is at most equal to the same object's score after the execution of s (as s' is a prefix of s), so if an object from R_{exact} is in R_s, it must be in $R_{s'}$ as well. But then the relative precision of s' is at least the relative precision of s, which is a contradiction. ■

Our algorithm to find the optimal sorted-access trace therefore considers all traces t where $t_i \in P_i$, for all i. This reasonably reduces the computational effort and makes an exact solution of this problem feasible at least for small values of k (up to 100) and m (up to 10).

B. Including Random Accesses

The algorithm for sorted-access traces can be extended to consider traces with sorted and random accesses. We make two important observations about random access in optimal traces: (1) random accesses occur always after sorted accesses have been finished, and (2) random access are only useful to objects in R_{exact}. To see (1), consider a trace with optimal relative precision that first makes some sorted accesses, then some random accesses, and then again some sorted accesses. The result of this schedule is identical to the result of a trace that first does all sorted accesses and then the random accesses. Regarding (2), assume that a random access was done to an object d not in R_{exact}. If this access (and other accesses to d) do not retrieve enough score to move d to the result, this access has no effect to the relative precision. Otherwise, d may replace a result from R_{exact} in the set of results, actually reducing relative recall. In both cases, relative recall can only get worse, not better, so this random access did not pay off.

Our extended algorithms now consider all sorted-access traces t (where $t_i \in P_i$) and for each trace t, as many additional random accesses as allowed by the remaining budget, i.e., $\left\lfloor \frac{1}{C_r} \cdot (B - C_s \cdot |t|) \right\rfloor$ random accesses. (Note that performing fewer random accesses will never lead to a better relative precision). We now start with the set R_t of results after the execution of t and refine it subsequently with random accesses. At any point during the execution of the algorithm, we denote with $mink$ the lowest score of any object in R_t. We maintain a priority queue Q of (object,list) pairs: For each seen object d in the trace t that is in R_{exact} and not in R_t, we compute the maximal score $best(d)$ d can get with a single RA; if this RA is to list L_i, we add the pair (d, L_i) with priority $best(d) - mink$ to Q. We remove the first pair (q, L_i) from Q, perform an RA to q in list L_i and update Q. We repeat this process until the budget is exhausted.

Note that this computation exploits the fact that we know the objects from R_{exact} and hence can schedule targeted RAs only for those results missing from the current R_t. Any real algorithm does not have this knowledge, so real relative precisions are usually much lower than the bound computed by this algorithm.

VII. Experiments

In this section, we present experimental results that demonstrate the top-k query processing under budget constraints. We use three data collections: data extracted from IMDB[1], the TREC Terabyte collection[2], and a synthetic collection.

The IMDB data has more than 375,000 movies and more than 1,200,000 persons (actors, directors, etc.). For each movie, we built a structured document with the attributes title, genre, actors, and description, and indexed the document with Lucene[3], resulting in one list for each (attribute,value)-pair. We used 20 queries from [5] as the benchmark. A query has four attributes: {Title, Genre, Actors, Description}. A typical query is Title="War" AND Genre="SciFi" AND Actors="Tom Cruise" AND Description="alien, earth, destroy".

The TREC Terabyte collection consists of more than 25 million Web pages from the .gov domain with a total size of about 426 gigabytes. We used 50 such queries and indexed the data in Lucene index using $TF \cdot IDF$ scoring as described in [2], resulting in a list of (document,score)- pairs for each term occurring in the collection. We chose 50 queries among the keyword queries provided with the collection (like "number of supreme court justices"), where the average length of the queries was three keywords.

The synthetic data was generated using a Zipfian distribution with controlled variations in the number of lists (2 to 6), number of objects (100000-1000000) and the parameters of the Zipfian distribution (exponent and rank).

To evaluate the results, we compare the result quality achieved by the different algorithms with the optimal quality that can be achieved by any budget-keeping algorithm, computed with the methods presented in Section VI. We used the following two measures:

- **percentage of optimal precision**: This measures how close the relative precision of the results achieved by an algorithm is to the optimal relative precision:

$$\frac{precision_{alg}}{precision_{opt}}$$

- **score mass error (SME)**: This reflects that even though the results of an algorithm may have little overlap with the exact results, they may still be equally good from an application point of view (such as different, but relevant text documents). Assuming that good documents have high aggregated scores, the deviation of the accumulated scores of the results from the accumulated scores of the exact results indicates the "goodness" of the results. Again we compare the score mass difference yielded by an algorithm with the score mass difference of the optimal solution:

$$SME = \frac{\sum_{i=1\ldots k} score_{exact}(i) - score_{alg}(i)}{\sum_{i=1\ldots k} score_{exact}(i) - score_{opt}(i)} \qquad (8)$$

  Here, $score_{exact}(i)$ denotes the aggregated score of the ith result in the exact results, $score_{opt}(i)$ in the optimal solution, and $score_{alg}(i)$ in the result of an algorithm.

We report macro-averages over the precision and SME values for all queries of the data set. We implemented the algorithms in Java 1.5 and ran the experiments on a machine with two Dual Intel Core 2.40GHz processors with 4MB cache each, an overall memory of 2GB, and a Linux Ubuntu operating system. In the following section we present our results and

[1]http://www.imdb.com/

[2]http://www-nlpir.nist.gov/projects/terabyte/

[3]http://lucene.apache.org/

provide analytical explanations for the different behavior of the algorithms.

A. Sorted Access Allocation

In this section we evaluate the different approaches for Sorted Accesses (SAs) using the different data sets. We compare the result of NRA [9], the Fair heuristic (Section IV-C), the Ranking heuristic (Section IV-D), and KBA [5] which is currently the state of the art top-k algorithm for sorted access only.

Figure 5 shows the percentage of the optimal precision for top-100 queries on the TREC dataset, varying the budget from 500 to 5000. It is evident that the Ranking and the Fair heuristics clearly outperform the NRA and KBA; in addition, the Ranking heuristic consistently achieves better results than the Fair heuristics.For instance, when the budget is 2000, the Ranking heuristic achieves 77% of the optimal precision, whereas Fair, KBA and NRA reach levels of 71%, 66% and 64%, respectively. On average, the Ranking achieves 78% of the optimal precision (which was 32% for a budget of 500 and 75% for a budget of 5000).

Table I shows the score mass errors for the different approaches in the same experiment, averaged over the different budgets. It is evident that our new Fair and Ranking heuristics produce results that are a lot closer to the optimal results. The results of Ranking are only 5% worse than the optimal results and will therefore probably satisfy the user as well.

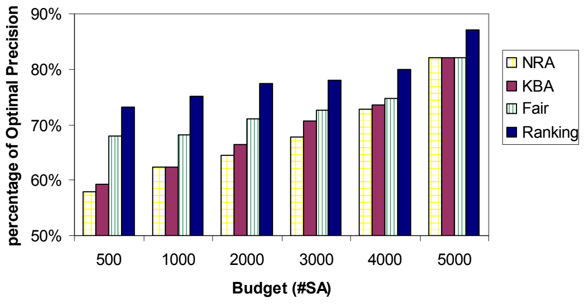


Fig. 5. TREC: average percentage of optimal precision, k=100, varying budgets

Algorithm	score mass error
NRA	1.31
KBA	1.24
$Fair$	1.16
$Ranking$	1.05

TABLE I

TREC: AVERAGE SCORE MASS ERROR (SME) , k=100

We turn now to the experiments on the IMDB collection. While the average length of the result lists for TREC is more than 1 million items, the IMDB result lists are very different. They usually consist of a mixture of long lists ($200,000$ items), short and very short (order of hundreds items) lists. For this reason, we limit the budget to be at most 1000 since usually after this number of accesses, the short lists are already completely accessed and all the different algorithms achieve the same results. Figure 6 shows the average percentage of the optimal precision for $k = 20$ of the different algorithms, varying the budget from 100 to 1000. The results are similar to the results on the TREC collection: again, the Ranking heuristic outperforms the rest, achieving an average of 65% of the optimal precision. Especially for very low budgets (between 200 and 500) Ranking is up to twice as good as NRA or KBA.

Table II shows the corresponding average score mass differences on IMDB for k=20 of the different approaches. Again, Ranking is very close to the optimal solution.

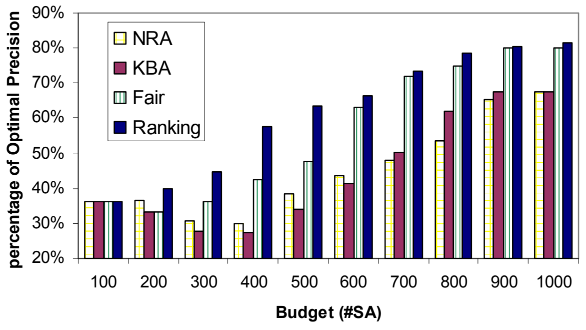


Fig. 6. IMDB: percentage of optimal precision for varying budget , k=20

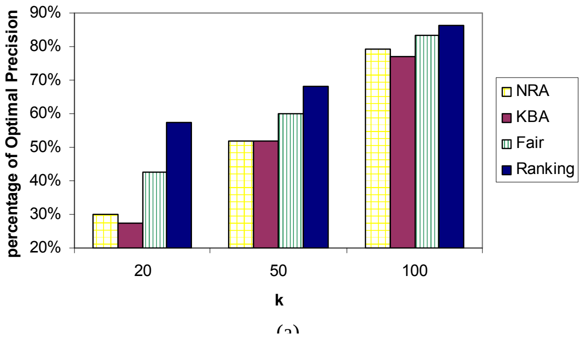
Algorithm	score mass error
NRA	1.406
KBA	1.822
$Fair$	1.266
$Ranking$	1.065

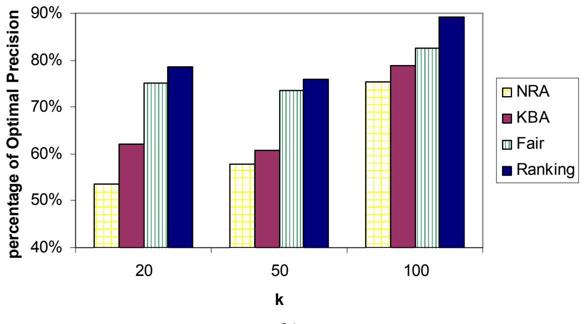
TABLE II

IMDB: AVERAGE SCORE MASS ERROR, k=20

Figures 7(a) and 7(b) vary the k value for budgets of 400 and 800, respectively. We can see that the algorithms behave similarly for larger values of k, even though the difference is less for larger k; All algorithms identify good result candidates, but Ranking is best at finding the very best results under low budget.

Finally, to check the effect of a different number of lists, we used the synthetic Zipfian distribution and varied the number of lists from 2 to 6, with $100,000$ objects in each list. We set the budget to be 4000 and compared the different approaches. Figure 8 summarizes the results. For the case of 2 lists, all approaches achieved 100% of the optimal precision. When the number of lists increases, the Ranking heuristic works very well and achieves improvements of around 20% over NRA. However, the gap between the Ranking heuristic and the optimal precision increases for large number of lists. This can be explained by the fact that for a small number of lists $(2, 3)$, there is less opportunity of scheduling different SAs so we can get close to the optimum. For a large number of lists $(5, 6)$, the algorithm has less chance to get enough knowledge on worst scores of candidates, thus uncertainty increases and precision decreases.





(b)

Fig. 7. IMDB: percentage of optimal precision for varying k (a) Budget = 400 (b) Budget = 800

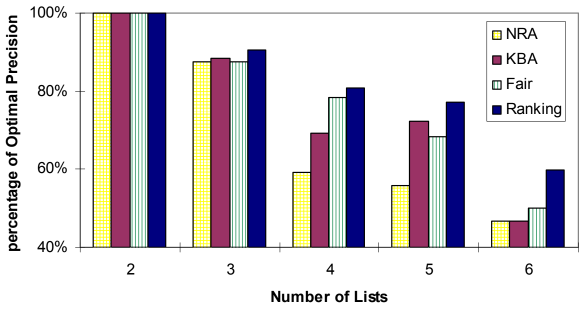


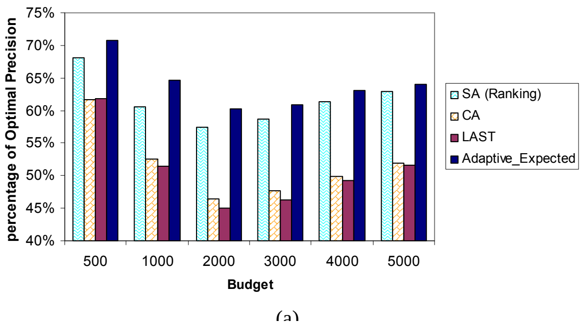
Fig. 8. Zipfian: percentage of optimal precision for varying number of lists, budget=4000 , k=100

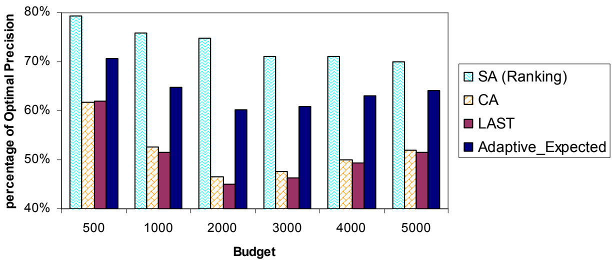
B. Switch to Random Access

In this section we evaluate the Adaptive Switch algorithm (denoted adaptive expected) under different budgets and access costs and compare it with the LAST algorithm [5], the CA algorithm [10], and the SA-only Ranking heuristic that does not perform any random accesses. Similarly to the experiments with sorted access only, we present percentages of the optimal precision and score mass errors. The experiments were done on the TREC and IMDB data sets. However, due to space limitations, we will report only the TREC results. The IMDB results were consistent with the TREC result.

Figures 9(a) and 9(b) show the percentage of the optimal precision for the TREC data set under different budgets varying from 500 to 5000 for $C_r = 10$ and $C_r = 100$, respectively. The first case shown in Figure 9(a) represents a situation where random accesses are cheap and therefore useful because we can perform quite a few of them before the budget runs out. Consequently, the Adaptive Switch reached a higher percentage of the optimal precision than the SA-only Ranking heuristics. The budget-oblivious CA and LAST algorithms perform much worse than our budget-aware algorithms, and worse still than a standard NRA would (as shown in the previous section). The Adaptive Switch achieves on average 64% of the optimal precision. Recall, however, that this optimal precision was achieved by an ideal algorithm that schedules only random accesses to relevant results; thus, it becomes very challenging for any algorithm to reach such precision levels. The results with the average score mass error (Table III) support the conjecture that random accesses are beneficial if they are relatively cheap.

We now increase the C_r to be 100, so random accesses are now much more expensive and we can usually afford only a few of them. Figure 9(b) shows that in this case, random accesses usually have a negative effect on result quality and should be avoided. A similar conclusion can be drawn from the average score mass error shown in Table IV.





(b)

Fig. 9. TREC: percentage of optimal precision for varying budgets, k=100 (a) $C_r = 10$ (b) $C_r = 100$

Algorithm	score mass error
Ranking	1.5943
CA	1.3905
LAST	1.7524
Adaptive_Expected	1.1788

TABLE III

TREC: AVERAGE SCORE MASS ERROR , k=100, C_r=10

VIII. CONCLUSIONS

The work presented here is the first attempt to deal with TA-style top-k query processing under budget constraints. The

Algorithm	score mass error
$Ranking$	1.2633
CA	1.5231
$LAST$	1.6222
$Adaptive\_Expected$	1.2794

TABLE IV
TREC: AVERAGE SCORE MASS ERROR , k=100, C_r=100

problem is very different in nature from the regular top-k problem. We presented a family of algorithms for sorted access only that achieve a precision between 65% and 77% of that of an optimal algorithm. We showed that random accesses could be beneficial if they are relatively cheap. However, when their cost increases, sorted-only algorithms are preferable. Future work can include the adaptation and extension of the model into a "time-aware" top-k query processing and considering different access costs for the different lists.

ACKNOWLEDGMENT

Michal Shmueli-Scheuer, Yosi Mass and Gerhard Weikum are partially funded by the European Commission Sixth Framework Programme project SAPIR. Chen Li is partially supported by the NSF CAREER Award No. IIS-0238586, a Google Research Award, and a gift fund from Microsoft. We thank Benjamin Sznajder and Doron Cohen for helping with indexing the collections, and Ronen Vaisenberg for useful discussions.

APPENDIX

A. NP-hardness proof

We reduce the 0-1 knapsack problem, which is known as an NP-Hard problem [11], to BSAS. We set α to be 0. Thus, we only consider the score reduction part. The 0-1 knapsack decision problem is defined as follows; Given m items X_i (i = $1 \dots m$), each with weight w_i and utility u_i, and a weight capacity C, find a subset S such that the total utility is maximized and the capacity constraint $\sum_{j\in S} w_j \leq C$ is satisfied. Given an instance of knapsack, we construct the following instance of we construct an instance of BSAS problem as follows. We consider m lists where the ith list has at its first $w_i - 1$ positions a constant score of 1 and thus score decrease 0, at position w_i a score decrease u_i (i.e., a resulting score 1-u_i, and subsequently the same constant score), i.e., no further score decrease. We claim that (A) a packing for this instance of knapsack has capacity $\leq C$ and utility $\geq U$ if and only if (B) the corresponding BSAS instance has a scan of total depth C and score decrease of $\geq U$. Proof of (A)$\Longrightarrow$ (B) : Given (A), we have $i_1, \dots, i_k$ such that $w_{i_1} + \dots + w_{i_k} \leq C$ and $u_{i_1} + \dots + u_{i_k} \geq U$. Then scanning lists $i_1, \dots, i_k$ to depths $w_{i_1} + \dots + w_{i_k}$, respectively, yields a total scan depth $\leq C$ (and we can get exactly C by scanning a few more positions without further score decrease in any of the lists) and a total score decrease of $u_{i_1} + \dots + u_{i_k} \geq U$. Proof of (B)$\Longrightarrow$(A) : Given (B), let $i_1, \dots, i_k$ be the lists where a non-zero score decrease has been achieved. List i_j has then been scanned at least to depth w_{i_j} , and therefore the total scan depth C is at least $w_{i_1} + \dots + w_{i_k}$. The total score reduction of these lists is exactly $u_{i_1} + \dots + u_{i_k}$, which by (B) is $\geq U$. The choice of items $i_1, \dots, i_k$ yields a packing that satisfies (A). $\Box$

REFERENCES

[1] A. Al-Hamdani and G. Özsoyoglu. Selecting topics for web resource discovery: Efficiency issues in a database approach. In *DEXA*, pages 792–802, 2003.

[2] E. Amitay, D. Carmel, and D. Cohen. Lucene and juru at trec 2007: 1-million queries track.

[3] B. Arai, G. Das, D. Gunopulos, and N. Koudas. Anytime measures for top-k algorithms. In *VLDB*, pages 648–659, 2007.

[4] W.-T. Balke, U. Güntzer, and W. Kießling. On real-time top k querying for mobile services. In *DOA/CoopIS/ODBASE*, pages 125–143, London, UK, 2002. Springer-Verlag.

[5] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[6] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.

[7] K. C.-C. Chang and S. won Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *SIGMOD*, 2002.

[8] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB*, pages 397–410, 1999.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, New York, NY, USA, 2001. ACM Press.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of Np-Completeness*. Freeman, San Francisco, 1979.

[12] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428, 2000.

[13] U. Güntzer, W.-T. Balke, and W. Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628, 2001.

[14] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. In *ACM Computing Surveys*, 2008. http://www.cs.uwaterloo.ca/∼ilyas/papers/IlyasTopkSurvey.pdf.

[15] A. Marian, N. Bruno, and L. Gravano. Evaluating top- queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[16] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, pages 22–29, 1999.

[17] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, pages 648–659, 2004.