IO-Top-k: Index-access Optimized Top-k Query Processing

Holger Bast Debapriyo Majumdar Ralf Schenkel Martin Theobald Gerhard Weikum

Max-Planck-Institut für Informatik, Saarbrücken, Germany
{bast,dmajumda,schenkel,mtb,weikum}@mpi-inf.mpg.de

ABSTRACT

Top-k query processing is an important building block for ranked retrieval, with applications ranging from text and data integration to distributed aggregation of network logs and sensor data. Top-k queries operate on index lists for a query's elementary conditions and aggregate scores for result candidates. One of the best implementation methods in this setting is the family of threshold algorithms, which aim to terminate the index scans as early as possible based on lower and upper bounds for the final scores of result candidates. This procedure performs sequential disk accesses for sorted index scans, but also has the option of performing random accesses to resolve score uncertainty. This entails scheduling for the two kinds of accesses: 1) the prioritization of different index lists in the sequential accesses, and 2) the decision on when to perform random accesses and for which candidates.

The prior literature has studied some of these scheduling issues, but only for each of the two access types in isolation. The current paper takes an integrated view of the scheduling issues and develops novel strategies that outperform prior proposals by a large margin. Our main contributions are new, principled, scheduling methods based on a Knapsack-related optimization for sequential accesses and a cost model for random accesses. The methods can be further boosted by harnessing probabilistic estimators for scores, selectivities, and index list correlations. In performance experiments with three different datasets (TREC Terabyte, HTTP server logs, and IMDB), our methods achieved significant performance gains compared to the best previously known methods.

1. INTRODUCTION

1.1 Motivation

Top-k query processing is a key building block for data discovery and ranking and has been intensively studied in the context of information retrieval [6, 21, 26], multimedia similarity search [10, 11, 12, 22], text and data integration [15, 18], business analytics [1], preference queries over product catalogs and Internet-based recommendation sources [3, 22], distributed aggregation of network logs and sensor data [7], and many other important application areas. Such queries evaluate search conditions over multiple attributes or text keywords, assign a numeric score that reflects the similarity or relevance of a candidate record or document for each condition, then combine these scores by a monotonic aggregation function such as weighted summation, and finally return the top-k results that have the highest total scores. The method that has been most strongly advocated in recent years is the family of *threshold algorithms (TA)* [12, 14, 25] that perform index scans over precomputed index lists, one for each attribute or keyword in the query, which are sorted in descending order of per-attribute or per-keyword scores. The key point of TA is that it aggregates scores on the fly, thus computes a lower bound for the total score of the current rank-k result record (document) and an upper bound for the total scores of all other candidate records (documents), and is thus often able to terminate the index scans long before it reaches the bottom of the index lists, namely, when the lower bound for the rank-k result, the *threshold*, is at least as high as the upper bound for all other candidates. Additionally, for promising candidates, unknown scores for some attributes can be looked up with random accesses, making the score bounds more precise.

When scanning multiple index lists (over attributes from one or more relations or document collections), top-k query processing faces an optimization problem: combining each pair of indexes is essentially an equi-join (via equality of the tuple or document ids in matching index entries), and we thus need to solve a join ordering problem [8, 15, 20]. As top-k queries are eventually interested only in the highest-score results, the problem is not just standard join ordering but has additional complexity. [15] have called this issue the problem of finding optimal rank-join execution plans. Their approach is based on a DBMS-oriented compile-time view: they consider only binary rank joins and a join tree to combine the index lists for all attributes or keywords of the query, and they generate the execution plan before query execution starts. An alternative, run-time-oriented, approach follows the Eddies-style notion of adaptive join orders on a per tuple basis [2] rather than fixing join orders at compile-time. Then the query optimization for top-k queries with threshold-driven evaluation becomes a *scheduling problem.* This is the approach that we pursue in this paper. In contrast to [2, 15] we do not restrict ourselves to trees of binary joins, but consider all index lists relevant to the query together.

List 1	**List 2**	**List 3**
Doc17 : 0.8	Doc25 : 0.7	Doc83 : 0.9
Doc78 : 0.2	Doc38 : 0.5	Doc17 : 0.7
·	Doc14 : 0.5	Doc61 : 0.3
·	Doc83 : 0.5	·
·	·	·
·	Doc17 : 0.2	·
·	·	·

Round 1 (SA on 1,2,3)

Doc17 :	[0.8 , 2.4]
Doc25 :	[0.7 , 2.4]
Doc83 :	[0.9 , 2.4]
unseen:	≤ 2.4

Round 2 (SA on 1,2,3)

Doc17 :	[1.5 , 2.0]
Doc25 :	[0.7 , 1.6]
Doc83 :	[0.9 , 1.6]
unseen:	≤ 1.4

Round 3 (SA on 2,2,3**!**)

Doc17 :	[1.5 , 2.0]
Doc83 :	[1.4 , 1.6]
unseen:	≤ 1.0

Round 4 (RA for Doc17)

Doc17 : 1.7
all others < 1.7
done!

Figure 1: A top-1 computation on three index lists, with three rounds of sorted access, followed by one round of random access.

The potential cost savings for flexible and intelligent scheduling of index-scan steps result from the fact that the descending scores in different lists exhibit different degrees of skew and may also be correlated across different lists. For example, dynamically identifying one or a few lists where the scores drop sharply after the current scan position may enable a TA-style algorithm to eliminate many top-k candidates much more quickly and terminate the query execution much earlier than with standard round-robin scheduling or the best compile-time-generated plan. These savings are highly significant when index lists are long, with millions of entries that span multiple disk tracks, and the total data volume rules out a solution where all index lists are completely kept in memory (i.e., with multi-Terabyte datasets like big data warehouses, Web-scale indexes, or Internet archives).

As an example for the importance of scheduling strategies, consider a top-1 query with three keywords and the corresponding index lists shown in Fig. 1. In the first two rounds, the first two documents from the top of the three lists are scanned, and lower and upper bounds on the final scores of the encountered documents are computed. At this point we have seen all potential candidates for the top document (because we know that the top document has a score of at least 1.5, while any document not yet encountered at all has a score of at most 1.4). However, if we stopped sorted accesses now, we might have to do up to *five* random accesses (one for Doc17, two for Doc25, and two for Doc83) to resolve which document has the highest score. In this situation a clever algorithm will opt to continue with sorted accesses. In the third round, now *two* documents from list 2 are scanned, one from list 3, and *none* from list 1. This is to bring down the threshold for unseen documents as much as possible and at the same time maximize the chance of encountering one of our candidate documents in a list where we have not yet seen it. In our example, this indeed happens: the threshold drops considerably, we no longer have to consider Doc25, and we get new information on Doc83. The algorithm now estimates that one more random access is likely to be enough to resolve the top document (because Doc17 is likely to get a better score than Doc83). It therefore stops doing sorted accesses and does a random access for Doc17 (the most promising in the example), after which the top document is indeed resolved and the algorithm can stop. The details of when our algorithms perform which kind of accesses on which lists and why are given in Sec. 4 and 5.

1.2 Problem Statement

The problem that we address in this paper is how to schedule index-access steps in TA-style top-k query processing in the best possible way, integrating sequential index scans and random lookups. Our goal is to minimize *the sum of the access costs*, assuming a fixed cost c_S for each sorted access and a fixed cost c_R for each random access. The same assumptions were made in [11]. We also study how to leverage statistics on score distributions for the scheduling of index-scan steps. The statistics that we consider in this context are histograms over the score distributions of individual index lists and also the correlations between index lists that are processed within the same query. For the prediction of aggregated scores over multiple index lists, we efficiently compute histogram convolutions at query run-time.

Throughout this paper, we assume that the top-k algorithm operates on precomputed index lists. We realize that this may not always be possible, for example, when a SQL query with a *stop-after* clause uses non-indexed attributes in the *order-by* clause. The latter situation may arise, for example, when expensive user-defined predicates are involved in the query [9, 10, 20] (e.g., spatial computations or conditions on images, speech, etc.). In these cases, the query optimizer needs to find a more sophisticated overall execution plan, but it can still use a threshold algorithm as a subplan on the subset of attributes where index lists are available. However, for text-centric applications and for semistructured data such as product catalogs or customer support, there is hardly a reason why the physical design should not include single-attribute indexes on all attributes that are relevant for top-k queries. Such application classes tend to be dominated by querying rather than in-place updates, and the disk space cost of single-attribute indexes is not an issue. The methods presented in this paper aim at such settings.

1.3 Related Work

The original scheduling strategy for TA-style algorithms is round-robin over all lists (mostly to ensure certain theoretical properties). Early variants also made intensive use of *random access (RA)* to index entries to resolve missing score values of result candidates, but for very large index lists with millions of entries that span multiple disk tracks, the resulting random access cost c_R is 50 - 50,000 times higher than the cost c_S of a *sorted access (SA)*. To remedy this, [12, 14] proposed the NRA (No RA) variant of TA, but occasional, carefully scheduled RAs can still be useful when they can contribute to major pruning of candidates. Therefore, [11] also introduced a *combined algorithm* (CA) framework but did not discuss any data- or scoring-specific scheduling strategies.

[14] developed heuristic strategies for scheduling SAs over multiple lists. These are greedy heuristics based on limited or crude estimates of scores, namely, the score gradients up to the current cursor positions in the index scans and the average score in an index list. This leads to preferring SAs on index lists with steep gradient [14].

[9] and [5, 22] developed the strategies MPro, Upper, and Pick for scheduling RAs on "expensive predicates". They considered restricted attribute sources, such as non-indexed attributes or Internet sites that do not support sorted access at all (e.g., a streetfinder site that computes driving distances and times), and showed how to integrate these sources into a threshold algorithm. [22] also considered sources with widely different RA costs or widely different SA costs (e.g., because of different network bandwidth or server load). Our computational model differs from these settings in that we assume that all attributes are indexes with support for both SA and RA and that all index lists are on the same server and thus have identical access costs. For our setting, MPro [9] is essentially the same as the Upper method developed in [5, 22].

Upper alternates between RA and SA steps. For RA scheduling, Upper selects the data item with the highest upper bound for its final score and performs a single RA on the attribute (source) with the highest expected score (with additional considerations to source-specific RA costs and eliminating "redundant" sources, which are not relevant here). This is repeated until no data item remains that has a higher upper bound than any yet unseen document could have; then SA are scheduled in a round-robin way until such a data item appears again. [5] also developed the Pick method that runs in two phases. In the first phase, it makes only SA until all potential result documents have been read. In the second phase, it makes RA for the missing dimensions of candidates that are chosen similarly to Upper.

Our own recent work [29] has used histograms and dynamic convolutions on score distributions to predict the total score of top-k candidates for more aggressive pruning; the scheduling in that work is standard round-robin, however. Probabilistic cost estimation for top-k queries has been a side issue in the recent work of [30], but there is no consideration of scheduling issues. Our TopX work on XML IR [28] included specific scheduling aspects for resolving structural path conditions, but did not consider the more general problem of integrated scheduling for SAs and RAs.

The RankSQL work [16, 20] considers the order of binary rank joins at query-planning time. Thus, at query run-time there is no flexible scheduling anymore. For the planning-time optimization, RankSQL uses simple statistical models, assuming that scores within a list follow a Normal distribution [16]. This assumption is made for tractability, to simplify convolutions. Our experience with real datasets indicated more sophisticated score distributions that are very different from Normal distributions, and we use more powerful statistics like explicit histograms with histogram convolutions computed at query time to deal with them.

1.4 Contribution

This paper makes several novel contributions:

- It develops novel strategies for sorted-access (SA) scheduling in TA-style top-k query processing that are based on a knapsack-related optimization technique.
- It develops a statistics-based cost model for random-access (RA) scheduling that employs statistical score predictors, selectivity estimators, and estimation of correlations among attribute values and/or keywords and provides an integrated strategy that combines SA and RA scheduling.
- It shows how these methods are best integrated into a high-performance top-k query engine that uses a combination of low-overhead merge joins with TA-style processing based on inverted block-index structures.
- It presents stress tests and large-scale performance experiments that demonstrate the viability and significant benefits of the proposed scheduling strategies.

On three different datasets (TREC Terabyte, HTTP server logs, and IMDB), our methods achieve significant performance gains compared to the best previously known methods, Fagin's Combined Algorithm (CA) and variants of the Upper and Pick [5, 22] algorithms: a factor of up to 3 in terms of abstract execution costs, and a factor of 5 in terms of absolute run-times of our implementation. We also show that our best techniques are within 20 percent of a lower bound for the execution cost of any top-k algorithm from the TA family; so we are fairly close to the optimum scheduling.

2. COMPUTATIONAL MODEL

2.1 Query and Data Model

We consider data items, structured records or text (or semistructured) documents, d_j ($j = 1 \dots N$), each containing a set of attribute values or keywords (terms) that spawn an M-dimensional Cartesian-product space. We associate with each record-value or document-term pair a numeric *score* that reflects the "goodness" or relevance of the data item with regard to the value or term. For example, for a price attribute of structured records, the score could be inversely proportional to the amount (cheaper is better), for a sensor the score could depend on the deviation from a target point (e.g., a desired temperature or the set value of a control parameter), and for text or semistructured documents the score could be an IR relevance measure such as TF·IDF or the probabilistic BM25 score derived from term frequencies (TF) and inverse document frequencies (IDF) [13]. We denote the score of data item d_j for the ith dimension by s_{ij}. Scores are often normalized to the interval $[0, 1]$, with 1 being the best possible score. We will assume such normalized scores in this paper, but this is not a critical assumption.

Top-k queries are essentially partial-match queries on the M-dimensional data space: $1 < m \leq M$ (usually $m \ll M$) conjunctions of primitive conditions of the form *attribute = value* or *document contains term*, but the conditions are interpreted as relaxable so that not matching one of them does not disqualify a candidate item for the query result and approximate matches are scored and ranked. Like most of the literature, we assume that the total score of an item is computed by a *monotonic score aggregation* function from the per-value or per-term scores of the item, e.g., using weighted summation. A top-k query returns k matches or approximate matches with the highest total scores.

2.2 Inverted Block-Index

The data items containing specific values or terms and their corresponding scores are precomputed and stored in "inverted" *index lists* L_i ($i = 1..M$), with one such list per data dimension (value or term). The entries in a list are `<itemID, score>` pairs. The lists may be very long (millions of entries) and reside on disk. We partition each index list into blocks and use score-descending order among blocks but keep the index entries within each block in `itemID` order. This is key to a low-overhead maintenance of the fairly extensive bookkeeping information that is necessary for TA-style query processing. We coin this hybrid structure *Inverted Block-Index*. The block size is a configuration parameter that is chosen in a way that balances disk seek time and transfer rate; a typical block size would be 32,768. More details on this index data structure and how we use it for high-performance query processing can be found in [4].

2.3 Query Processing

Our query processing model is based on the NRA and CA variants of the TA family of algorithms. An m-dimensional top-k query (with m search conditions) is primarily processed by scanning the corresponding m index lists in descending score orders in an interleaved manner (and by making judicious random accesses to look up index entries of specific data items). Without loss of generality, we assume that these are the index lists numbered L_1 through L_m. For numerical or categorical attribute-value conditions that are not perfectly matched, the query processor considers "alternative" values in ascending order of similarity to the original value of the query (thus preserving the overall descending-score processing order). For example, when searching for $year = 1999$, after exhausting the index list for the value 1999, the next best lists are those for 1998, 2000, etc. Although this relaxation involves additional lists, we treat this procedure as if it were a single index scan where the list for 1999 is conceptually extended by "neighboring" lists.

When scanning the m index lists, the query processor collects candidates for the query result and maintains them in two priority queues, one for the *current top-k items* and another one for *all other candidates* that could still make it into the final top-k. For simpler presentation, we assume that the score aggregation function is simple summation. The query processor maintains the following state information:

- the current cursor position pos_i for each list L_i,
- the score values $high_i$ at the current cursor positions, which serve as upper bounds for the unknown scores in the lists' tails,
- a set of current top-k items, d_1 through d_k (renumbered to reflect their current ranks) and a set of data items d_j $(j = k + 1..k + q)$ in the current candidate queue Q, each with
  - a set of evaluated dimensions $E(d_j)$ in which d_j has already been seen during the scans or by random lookups,
  - a set of remainder dimensions $\bar{E}(d_j)$ for which the score of d_j is still unknown,
  - a lower bound $worstscore(d_j)$ for the total score of d_j which is the sum of the scores from $E(d_j)$,
  - an upper bound $bestscore(d_j)$ for the total score of d_j which is equal to
$$worstscore(d_j) + \sum_{\nu \in \bar{E}(d_j)} high_\nu$$

In addition, the following information is derived at each step:

- the minimum worstscore *min-k* of the current top-k docs, which serves as the stopping threshold,
- the bestscore that any currently unseen document can get, which is computed as the sum of the current $high_i$,
- and for each candidate, a score deficit $\delta_j = min\text{-}k - worstscore(d_j)$ that d_j would have to reach in order to qualify for the current top-k.

The top-k queue is sorted by worstscore values, and the candidate queue is sorted by descending bestscore values. Ties among scores may be broken by using the concatenation of `<score, itemID>` for sorting. The invariant that separates the two is that the rank-k worstscore of the top-k queue is at least as high as the best worstscore in the candidate queue. The algorithm can safely terminate, yielding the correct top-k results, when the maximum bestscore of the candidate queue is not larger than the rank-k worstscore of the current top-k, i.e., when

$$\min_{d \in top\text{-}k}\{worstscore(d)\} =: min\text{-}k \geq \max_{c \in Q}\{bestscore(c)\}$$

More generally, whenever a candidate in the queue Q has a bestscore that is not higher than *min-k*, this candidate can be pruned from the queue. Early termination (i.e., the point when the queue becomes empty) is one goal of efficient top-k processing, but early pruning to keep the queue and its memory consumption small is an equally important goal (and is not necessarily implied by early termination). The candidate bookkeeping is illustrated in Fig. 2.

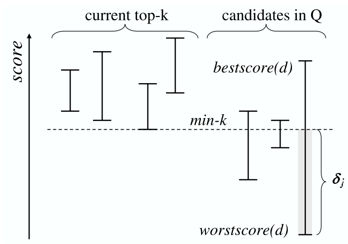

Figure 2: Top-k and candidate bookkeeping.

In the rest of the paper we will primarily use the IR-oriented terminology of documents and terms. It is straightforward to carry over our methods and results to settings with numerical or categorical attributes of structured records.

2.4 Taxonomy of Scheduling Strategies

Different algorithm within the TA-sytle family differ in the ways how they handle three fundamental issues: (1) how SAs are scheduled, (2) how RAs are scheduled, and (3) how RAs are ordered. This section presents a taxonomy of the different possibilities for each dimension, classifies the existing approaches in this scheme and points out the new approaches presented in this paper.

2.4.1 SA-Scheduling

`RR`: Schedule SA in a round-robin manner across the lists (TA, NRA, CA, Upper, and Pick).

`KSR`: Schedule different amounts of SA per list in order to maximize the reduction of scores at the future scan positions for a fixed batch of sorted accesses, using a Knapsack-based optimization algorithm (see Sec. 4.1).

`KBA`: Schedule different amounts of SA per list in order to maximize an aggregated benefit among all candidates currently being in the queue for a fixed batch of sorted accesses, using a Knapsack-based optimization algorithm (see Sec. 4.2).

2.4.2 RA-Scheduling

`Never`: Perform SA only (NRA).

`All`: After each round of SA, perform full RA for each new candidate to retrieve its final score; no candidate queuing is required (TA).

`Top`: After each round of SA, schedule RA on the currently best candidates in the queue (including the not yet fully evaluated documents currently in the top-k). An extreme instance of such an algorithm is Upper that schedules RA for all candidates that have a higher bestscore than the current bestscore of a yet unseen document.

`Each`: After each round of SAs, schedule a balanced amount of RA according to the current cost ratio c_R/c_S between RA and SA performed so far (CA).

`Last`: Perform only batches of SAs initially and, at some point in the algorithm, switch to performing only RA, thus scheduling the full amount of RA to eliminate all the remaining items in the queue. One algorithm in this class is Pick that switches from SA to RA as soon as the best score that an unseen document can get drops below the current *min-k* threshold. In contrast, our algorithms (see Sec. 5.1 and 5.2) stop the SA batches according to the *estimated* cost for the *remaining* RA (i.e., corresponding to the estimated number of candidates in the queue that need to be looked up to raise *min-k* above the bestscore of the currently best candidate).

2.4.3 RA-Ordering

`Best`: Perform RAs in descending order of $bestscore(d_j)$ (CA, Upper and Sec. 5.1).

`Ben`: Perform RAs according to a cost model, i.e., proportionally to the probability $p(d_j)$ that d_j gets into the top-k results (see Sec. 5.2).

Any algorithm for TA-style top-k query processing now corresponds to a triplet, for example, the NRA scheme corresponds to `RR-Never`, TA corresponds to `RR-All`, CA is `RR-Each-Best`, and Upper corresponds to `RR-Top-Best`. In Sections 4 and 5, we will investigate the more sophisticated combinations. Our best results will be obtained by the combination `KSR-Last-Ben`.

2.5 Computing Lower Bounds

[12] proved that the CA algorithm has costs that are always within a factor of $4m + k$ of the optimum, where m is the number of lists and k is the number of top items we want to see. Even for small values of m and k, this factor is fairly large (e.g., 22 if we want the top-10 of a 3-word query), and, it seems, way too pessimistic. [5] presented a way to compute a lower bound on the cost of *individual* queries for the special case of queries with only a single indexed attribute We extend their approach to our setting where all lists can be accessed with sorted and random accesses. For any top-k query processing method, after it has done its last SA, consider the set X of documents which were seen in at least one of the sorted accesses, and which have a bestscore not only above the current *min-k* score, but even above the final *min-k* score (which the method does not know at this time). If only a fraction of each list has been scanned, this set X is typically of considerable size. Now it is not hard to see that the method *must do an RA for every document from* X in order to be correct.

Therefore, the following construction gives a lower bound on the cost of top-k method: try all possible combinations of scan depths in each of the input lists, and for each such combination compute the cost of scanning until this depth plus the cost of the then absolutely necessary RAs. In this computation, we restrict ourselves to scan depths that are multiples of a certain block size. Note that the outlined computation is not a real top-k algorithm itself, but merely serves to determine lower bounds for comparison.

3. PROBABILISTIC FOUNDATIONS

3.1 Score Predictor

In this section we develop the details for estimating the probability $p(d)$ that a candidate document d with non-empty remainder set $\bar{E}(d)$ may qualify for the top-k results. The way how we estimate $p(d)$ depends on the assumptions that we make about the distribution of the unknown scores that d would obtain from each remaining list; for each missing dimension, we consider a random variable S_i for the score of d in that dimension. As we don't know the actual distribution of the S_i unless we have read the whole list, we have to model or approximate the distribution. We use histograms [17] as an efficient and commonly used means to compactly capture arbitrary score distributions. In our application, we precompute a histogram for the score distribution of each index list, discretizing the score domain for each index list into H buckets with lower buckets bounds $s_1, \ldots, s_H$ and storing the respective document frequency and the cumulated document frequency for each of the histogram buckets. Using the approximated distributions of the scores in each list, we can estimate the probability that a candidate document can get enough score mass from its remaining lists to enter the top-k as

$$p_s(d_j) := P\left[\sum_{i\in\bar{E}(d_j)} S_i > \delta_j | S_i \le high_i\right]$$

As this involves the sum of random variables, we need to compute the convolution of the corresponding distributions to compute this probability. Our previous paper [29] has shown how to do this efficiently at query run-time. Among the techniques presented there, the current paper adopts histogram convolutions, which are recomputed periodically after every batch of SA steps. The computational overhead for the convolutions was never a bottleneck in the overall top-k algorithm.

3.2 Selectivity Estimator

The score predictor implicitly assumes that a document occurs in all its missing dimensions, hence it inherently overestimates the probability that a document can get a score higher than the current *min-k*. For a more precise estimation, we take the selectivity of the lists into account, i.e., the probability that a document occurs in the remaining part of a list. For a single list L_i with length l_i and a total dataset size of n documents, this probability is

$$q_i(d) := \frac{l_i - pos_i}{n - pos_i}$$

For a partially evaluated document d with a set $\bar{E}(d)$ of remainder dimensions, the probability $q(d)$ that d occurs in at least one of the dimensions in $\bar{E}(d)$ is computed as

$$\begin{aligned} q(d) &:= P[d \text{ occurs in at least one list in } \bar{E}(d)] \\ &= 1 - P[d \text{ does not occur in any list in } \bar{E}(d)] \\ &= 1 - \prod_{i\in\bar{E}(d)} (1 - q_i(d)) \end{aligned}$$

assuming independence for tractability. This independence assumption can be relaxed by the covariance-based technique mentioned in Sec. 3.4.

3.3 Combined Score & Selectivity

We write $A(d, Y')$ for the probabilistic event that d occurs in all lists Y' and in none of the remaining lists in $\bar{E}(d) \setminus Y'$, and $O(d, \bar{E}(d))$ for the probabilistic event that d occurs in at least one of the dimensions in $\bar{E}(d)$. Then the combined probability that a document d can reach the top-k can be estimated as follows:

$$\begin{aligned}
p(d) &:= P[d \in \textit{top-k}] \\
&= \sum_{Y' \subseteq \bar{E}(d)} P[A(d, Y') \wedge \sum_{i \in Y'} S_i > \textit{min-k}] \\
&\leq \sum_{Y' \subseteq \bar{E}(d)} P[A(d, Y') \wedge \sum_{i \in \bar{E}(d)} S_i > \textit{min-k}] \\
&= P[O(d, \bar{E}(d)) \wedge \sum_{i \in \bar{E}(d)} S_i > \textit{min-k}] \\
&= P[\sum_{i \in \bar{E}(d)} S_i > \textit{min-k} | O(d, \bar{E}(d))] \cdot \ P[O(d, \bar{E}(d))] \\
&= p_s(d) \cdot q(d)
\end{aligned}$$

This corresponds to a conjunctive combination of the probabilities from the score predictor and selectivity estimates.

3.4 Feature Correlations

Assuming that documents occur independently in different lists may lead to a crude and practically useless estimator as terms used in queries are frequently highly correlated. To capture this in our probability estimator, we precompute pairwise term covariances for terms in frequent queries (e.g., derived from query logs). For two such terms and their corresponding lists L_i and L_j, we use a contingency table to capture co-occurrence statistics for these terms. We denote by l_i the length of list L_i and by l_{ij} the number of docs that are in both L_i and L_j. We then consider the random variable X_i which is 1 if some doc d is in L_i (the same distribution for all d, but not the same value, of course), and 0 otherwise. To predict $X_j(d)$ after knowing $X_i(d) = 1$, we have to compute the covariance $cov(X_i(d), X_j(d))$ of X_i and X_j. Following basic probability theory, we can estimate this covariance as $cov(X_i, X_j) = \frac{l_{ij}}{n} - \frac{l_i \cdot l_j}{n^2}$.

We show how feature correlations can be exploited for a better estimation of selectivities. We want to estimate the probability $q_i(d)$ that a document d occurs in the remainder of the list L_i given that it already has occurred in some lists $E(d)$, using the pairwise covariances of L_i with the lists in $E(d)$. First we consider the case where $E(d) = \{j\}$ consists of a single list. Using the equality $P[X_i \wedge X_j] = P[X_i]P[X_j] + cov(X_i, X_j)$ for Bernoulli random variables, we can derive

$$\begin{aligned}
P[X_i|X_j] &= \frac{P[X_i \wedge X_j]}{P[X_j]} \\
&= \frac{P[X_i]P[X_j] + cov(X_i, X_j)}{P[X_j]} \\
&= \frac{\frac{l_i}{n} \cdot \frac{l_j}{n} + \frac{l_{ij}}{n} - \frac{l_i \cdot l_j}{n^2}}{\frac{l_j}{n}} = \frac{l_{ij}}{l_j}
\end{aligned}$$

We would like to estimate $P[X_i = 1|E(d)] := P[X_i = 1|X_1 = 1, X_2 = 1, ..., X_j = 1]$ with $E(d) = \{1, 2, \ldots, j\}$ and the elements of $E(d)$ conveniently renumbered. As we only have pairwise covariance estimates, we work with the approximation $P[X_i = 1|E(d)] \geq \max_{j \in E(d)} P[X_i = 1|X_j = 1]$ which yields

$$\begin{aligned}
q_i(d) &= P[X_i = 1|E(d)] \\
&\geq \max_{j \in E(d)} P[X_i = 1|X_j] \ = \ \max_{j \in E(d)} \frac{l_{ij}}{l_j}
\end{aligned}$$

We can plug this correlation-aware estimation for the probability that a document occurs in a single list in the selectivity estimator from Sec. 3.2 and the combined score predictor from Sec. 3.3.

4. SORTED ACCESS SCHEDULING

Index lists are processed in batches of b sorted accesses. That is, the query engine fetches b index entries from all m query-relevant index lists, and these b entries can be distributed arbitrarily across the lists. The priority queue Q for result candidates is rebuilt with updated priorities after each round of b such steps. For our inverted block index, as we described it in Sec. 2.2, we choose b as a multiple of the block size. Since the blocks are sorted by item IDs, the required bookkeeping can then be efficiently implemented via merge joins, without the need for any priority queue or hash data structure. Note that this implies a slightly coarser granularity of the TA-style query processing.

Our overriding goal is to minimze the weighted sum of sorted-access (SA) and random-access (RA) steps for computing the top-k results of a query: $c_S \times \#SA + c_R \times \#RA$. In this section, we assume a fixed strategy for RAs (e.g., no RAs at all in an NRA-style method or periodic RAs for the best candidates after every c_R/c_S rounds of SAs), and focus on the SA cost part. Our goal in SA scheduling is to optimize the individual batch sizes b_i ($i = 1..m$) across all the lists, i.e., choose $b_1, \ldots, b_m$ so as to maximize some benefit function under the constraint $\sum_{i=1}^{m} b_i = b$. For the block-organized index, the units of the scheduling decisions are entire blocks. In the following we will present our methods in terms of SAs to individual index entries; the block-oriented variant follows in a straightforward manner.

Inspired by the earlier work on simple scheduling heuristics [14], our first method aims to reduce the scores at the index scan positions, the $high_i$ bounds, as quickly as possible. The rationale of this strategy is that low $high_i$ values result in lower bestscores of *all* top-k candidates, which in turn enables us to prune more candidates earlier. It turns out, however, that this strategy does not perform well in many cases. We have developed a more general and typically better performing scheduling strategy that considers an explicit notion of *benefit* of a candidate in Q and aggregates over all candidates for a judicious decision on the b_i steps. The benefit function will be defined so as to strive for low SA costs in the overall objective function (weighted sum of SAs and RAs). Both strategies lead to the NP-hard knapsack problem, hence we have coined them KSR (Knapsack scheduling for Score Reduction) and KBA (Knapsack scheduling for Benefit Aggregation).

4.1 Knapsack for Score Reduction (KSR)

Given the current scan positions $pos_1, \ldots, pos_m$, we are looking for a schedule of $b_1, \ldots, b_m$ steps (with $b_1 + \cdots + b_m = b$), such that we maximize the total reduction in bestscores of the documents currently present in our candidate queue. For a candidate document $d \in Q$, $bestscore(d)$ reduces by $\Delta_i = high_i - score_i(pos_i + b_i)$ if $i \in \bar{E}(d)$ and by 0 if $i \in E(d)$ when we scan b_i elements further into list L_i and do not see the document d in the list L_i. Since the probability of seeing a particular document by scanning a small part of a list is close to zero, the expected reduction in $bestscore(d)$ can be considered as Δ_i. Hence the expected aggregated reduction in bestscores for all documents in Q is given by $w_i \Delta_i$ where $w_i = |\{d \in Q | i \in \bar{E}(d)\}|$ is the number of documents for which a reduction in bestscore is expected by scanning into list L_i. We can easily estimate the $score_i(pos_i + b_i)$ from the precomputed histograms, assuming a uniform distribution of scores within a histogram cell. We can now define our objective function for the choice of b_i values: maximize the score reduction $SR(b_1, \ldots, b_m) = \sum_{i=1}^{m} w_i \Delta_i$, where we treat the Δ_i values as a (deterministic) function of the b_i choices (ignoring potential estimation errors caused by the histograms). This problem is NP-hard, as we can reduce the well-known knapsack problem to it (see [4]). However, in all our applications the number m of lists is relatively small and we schedule only a small multiple of m in every round, so that we can actually check all possible combinations in very little time compared to the reads and merges.

4.2 Knapsack for Benefit Aggregation (KBA)

The knapsack scheduling framework introduced in the previous subsection is intriguing and powerful, but solely aiming to reduce the scores at the scan positions as quickly as possible is not the best optimization criterion. It allows us to identify some low-scoring candidates and prune them earlier, but it does not necessarily lead to more information about the high-scoring candidates. We do not only want to reduce the bestscore bounds of some candidates as much as possible, but are actually more concerned about the bestscore bounds of those candidates that are close to the *min-k* threshold and, more generally, would prefer a modest bestscore reduction of many candidates over a big reduction for some smaller fraction only. To address these issues we now define an explicit formalization of the *benefit* that we obtain from scanning forward by $(b_1, \ldots, b_m)$ positions in the m index lists, taking into consideration not only the current scan positions and score statistics, but also the knowledge that we have compiled about the documents seen so far during the scans. Benefit will be defined for each document, and we will then aggregate the benefits of all documents in the current top-k or the candidate queue Q.

Observe that if a candidate document d has already been seen in list L_i, then neither $bestscore(d)$ nor $worstscore(d)$ changes when we scan L_i further. So, for each list L_i, we shall consider only the documents $d \in Q$ which are not seen in L_i, i.e. $i \in \bar{E}(d)$. The probability $q_i^{b_i}(d)$ that a document d is seen in L_i in the next b_i steps is

$$\begin{aligned} q_i^{b_i}(d) &= P[d \text{ in next } b_i \text{ elements of } L_i | E(d)] \\ &= P[d \text{ in next } b_i | d \in L_i \wedge E(d)] \cdot P[d \in L_i | E(d)] \\ &= \frac{b_i}{l_i - pos_i} \cdot P[d \in L_i | E(d)] \\ &= \frac{b_i}{l_i - pos_i} \cdot P[X_i = 1 | E(d)] \\ &\geq \frac{b_i}{l_i - pos_i} \cdot \max_{j \in E(d)} \frac{l_{ij}}{l_j} \end{aligned}$$

If a document d is actually found in L_i by scanning further to depth b_i, the worstscore of d increases, which in turn contributes to increasing *min-k* and thus pruning more documents. The expected *gain* in $worstscore(d)$ when list L_i is scanned further to depth b_i is given by $q_i^{b_i}(d)\mu(pos_i, b_i)$ where $\mu(pos_i, b_i)$ is the mean score of the documents from current scan position pos_i to $pos_i + b_i$. We can estimate $\mu(pos_i, b_i)$ as well from the precomputed histogram. Similarly, we can estimate the reduction in bestscore of a candidate document $d \in Q$ with regard to list L_i as $(1 - q_i^{b_i}(d))\Delta_i$, if it is not seen in the next b_i steps. Now we can define our benefit function for every candidate document $d \in Q$ not already seen in list L_i as

$$\text{Ben}_i(d, b_i) = q_i^{b_i}(d)\mu(pos_i, b_i) + (1 - q_i^{b_i}(d))\Delta_i$$

and the total benefit of scanning to depth b_i in L_i as

$$\text{Ben}_i(b_i) = \sum_{d \in Q, i \in \bar{E}(d)} \text{Ben}_i(d, b_i)$$

Finally, we can define the overall benefit for a schedule $s = (b_1, \ldots, b_m)$ by a simple benefit aggregation:

$$\text{Ben}(s) = \sum_{i=1}^{m} \text{Ben}_i(b_i)$$

So we are looking for a schedule s for which the benefit $\text{Ben}(s)$ is maximized. This notion of overall benefit includes an implicit weighting of lists, by giving higher weight to the lists for which we have many documents in the queue that have not yet been seen there and which would benefit from a significant reduction of the $high_i$ bounds for these lists. Thus scanning on these lists could make the decisive difference between pruning many candidates or having to keep them in the queue.

5. RANDOM ACCESS SCHEDULING

Random-access (RA) scheduling is crucial both in the early and the late stages of top-k query processing. In the early stage, it is important to ensure that the *min-k* threshold moves up quickly so as to make the candidate pruning more effective as the scans proceed and collect large amounts of candidates. Later, it is important to avoid that the algorithm cannot terminate merely because of a few pieces of information missing about a few borderline candidates. In the following we present various strategies for deciding when to issue RAs and for which candidates in which lists. Some of them have a surprisingly simple heuristic nature, others are cost-model-driven. Following the literature [9, 22], we refer to score lookups by RA as *probing*.

5.1 Last-Probing

In *Last-Probing*, just as in CA, we do a balanced number of RAs, that is, we see that the total cost of the RAs is about the same as the total cost of all SAs. In CA, this is trivially achieved by doing one random access after each round of $\lceil c_R/c_S \rceil$ SA. In Last-Probing, we perform RAs only after the last round, that is, we have a phase of only SAs, followed by a phase of only RAs.

We do this by estimating, after each round, the number of RAs that would have to be done if this were the last round of SAs. Two criteria must be met for this round of SA being the last. First, the estimated number of RA must be less than $\lceil c_R/c_S \rceil$ times the number of *all* SA done up to this point Second, we must have $\sum_{i=1}^{m} high_i \leq$ *min-k*, since only then we can be sure that we have encounterd all the top-k items already. We remark that in all our applications, the second criterion is typically fulfilled long before (that is, after much fewer rounds than) the first criterion.

A trivial estimate for the number of random lookups that would have to be done if we stopped doing SAs at a certain point, is the number of candidate documents which are then in our queue. Clearly, this estimate is an upper bound. When the distribution is very skewed, it is in fact quite a good estimate, because then each document in the queue has a positive but only very tiny probability of becoming one of the top-k items. For more uniform score distributions like BM25, however, it turns out than we can do much better.

Consider the queue after some round, and assume an ordering of the documents by descending bestscores, i.e., highest bestscore first. For the ith document in that ordering, let W_i and B_i denote its worstscore and bestscore, respectively, and by F_i its final score (which we do not know before doing random accesses, unless $W_i = B_i$). Now consider the lth document (in the bestscore ordering), and let k' be the number of top-k items with worstscore below B_l. Then it is not hard to see that there will be a random lookup for this lth document, if and only if at most k' of the $l-1$ documents $d_1, \ldots, d_{l-1}$ have a final score larger than B_l. Let R_l be the random indicator variable that is 1 if that happens and 0 otherwise. Let $p_{i,l} := P[F_i > B_l]$, which can be computed as described in Sec. 3.1. Since the $p_{i,l}$ are small, and l tends to be large, the number of i for which $F_i > B_l$ can be approximated very accurately by a random variable X_l with a Poisson distribution with mean $p_{1,l} + \cdots + p_{l-1,l}$. We then have $E(R_l) = P[R_l = 1] = P[X_l < k]$, which can be computed very efficiently and accurately by means of the incomplete gamma function [27].

As described so far, the probabilities $p_{1,l}, \ldots, p_{l-1,l}$ would have to be computed from scratch for every document. The time for computing $\sum_l E(R_l)$ as an estimate for the number of RA would then be quadratic in the number of documents in the queue. We improve on this by approximating $p_{i,l} = P[F_i > B_l]$ by

$$\tilde{p}_{i,l} = P[F_i > \mathit{min\text{-}k}] \cdot \frac{B_l - \mathit{min\text{-}k}}{B_i - \mathit{min\text{-}k}}$$

Note that by the bestscore ordering we have that $B_l \le B_i$, for $i < l$. It then suffices to compute $P[F_i > \mathit{min\text{-}k}]$, once for each document i, and to maintain, while processing the documents in order of descending bestscores, the number of top-k items which are smaller than the current document, which can be done in linear overall time. It is not hard to see, that from these quantities, $\sum_{i=1}^{l-1} \tilde{p}_{i,l}$ can be computed in constant time, for any given l.

When doing the random accesses, it plays a role in which order we process the documents for which we do random lookups. In the basic Last-Probing, we simply order them by decreasing bestscore (`Last-Best`); this is similar to CA, which after each round of sorted accesses does an RA for the candidate document with the highest bestscore. In the next section, we see a more sophisticated ordering.

5.2 Ben-Probing

The *Beneficial Probing* strategy, *Ben-Probing* for short, extends the Last-Probing by a probabilistic cost model for assessing the benefit of making RAs versus continuing with SAs in the index scans. The cost comparison is updated periodically every b steps, i.e., whenever we need to make an SA-scheduling decision anyway. The cost is computed for each document d in the candidate queue or the current top-k separately; obviously SA costs per document are then fractions of the full SA costs as index-scan steps are amortized over multiple documents. Then we can either schedule RAs for individual documents based on the outcome of the cost comparison, or we can batch RAs for multiple candidates and would then simply aggregate the per-candidate RA costs. In the following, we first develop the cost estimates and then come back to the issue of specific scheduling decisions. We denote the number of documents in the priority queue by $q = |Q|$.

For both cost categories, we consider the *expected wasted cost (EWC)* which is the expected cost of random (or sorted) accesses that our decision would incur but would not be made by an optimal schedule that would make random lookups only for the final top-k and traverse index lists with minimal depths. To compute the EWCs, we set the cost of an SA to 1 and the cost of an RA to c_R/c_S, hence the model uses only the cost ratio, not the actual costs.

For looking up unknown scores of a candidate document d in the index lists $\bar{E}(d)$, we would incur $|\bar{E}(d)|$ RA which are wasted if d does not qualify for the final top-k result. We can compute this probability using the combined score estimator from Sec. 3.3 and exploiting correlations as shown in Sec. 3.4, as

$$\begin{aligned} P\left[d \notin \mathit{top\text{-}k}\right] &= 1 - p(d) \\ &= 1 - p_S(d) \cdot q(d) \\ &= 1 - p_S(d) \cdot \left(1 - \prod_{i \in \bar{E}(d)} (1 - q_i(d))\right) \\ &\le 1 - p_S(d) \cdot \left(1 - \prod_{i \in \bar{E}(d)} \left(1 - \max_{j \in E(d)} \frac{l_{ij}}{l_j}\right)\right) \end{aligned}$$

Then the random accesses to resolve the missing scores have expected wasted cost:

$$EWC_{RA}(d) \;:=\; |\bar{E}(d)| \cdot (1 - p(d)) \cdot \frac{c_R}{c_S}$$

Analogously, the next batch of b SA for an additional depth b_i at index list L_i, with $\sum_i b_i = b$, incurs a fractional cost to each candidate in the priority queue, and these total costs are shared by all $|Q|$ candidates. For a candidate d, the SA are wasted if either we do not learn any new information about the total score of d (that is, when we do not encounter d in any of the m remainder dimensions), or if we encounter d, but it does not make it to the top-k. Denoting the probability of seeing d in the i^{th} list in the next b_i steps as $q_i^{b_i}(d)$ like in Sec. 4.2, we can compute the probability $q^b(d)$ of seeing d in at least one list in the batch of size b as

$$\begin{aligned} q^b(d) &:= 1 - P[d \text{ not seen in any list}] \\ &= 1 - \prod_{i \in \bar{E}(d)} (1 - P[d \text{ seen in } L_i \text{ in next } b_i \text{ steps}]) \\ &= 1 - \prod_{i \in \bar{E}(d)} \left(1 - q_i^{b_i}(d)\right) \end{aligned}$$

Hence the probability of *not* seeing d in any list is $1 - q^b(d)$. The probability that d is seen in at least one list, but does not make it to the top-k can be computed as $q_S(d) := (1 - p_S(d)) \cdot q^b(d)$ analogously to Sec. 3.3. Then the total costs for the next batch of b SA are shared by all candidates in Q, and this incurs expected wasted cost:

$$\begin{aligned} EWC_{SA} &:= \frac{b}{|Q|} \cdot \sum_{d \in Q} \Big((1 - q^b(d)) + (1 - p_S(d)) \cdot q^b(d) \Big) \\ &= \frac{b}{|Q|} \cdot \sum_{d \in Q} \Big(1 - q^b(d) \cdot p_S(d) \Big) \end{aligned}$$

We can now replace the real costs (as counted by Last-Probing) with the expected wasted costs EWC_{RA} and EWC_{SA} for the Ben-Probing. In order to trigger random accesses for specific candidates, we always consider the *cumulated* EWC_{RA} costs and compare them to the *cumulated* EWC_{SA} of all batches done so far. For `Last-Ben`, we exclusively perform SA batches until the sum of the expected wasted costs of all remaining candidates in the queue is less than the cumulated expected wasted costs of all previous SA batches; we then perform the RAs for all documents in the queue in ascending order of the documents' EWC_{RA}.

For each candidate d, we actually perform the RAs one at a time in ascending order of index list selectivity l_i/n, for all $i \in \bar{E}(d)$, thus counting a single RA for each candidate and list. We may safely break this sequence of RAs on d, if $bestscore(d) \leq min\text{-}k$, hence drop that candidate, and save some RA costs for another candidate.

6. EXPERIMENTS

6.1 Data Collections & Setup

We consider three structurally different data collections: the TREC Terabyte collection, movie data from IMDB, and a huge HTTP server log. The TREC Terabyte benchmark collection[1] consists of more than 25 million Web pages from the .gov domain, mostly HTML and PDF files with a total size of about 426 gigabytes. It provides a set of 50 keyword queries like "kyrgyzstan united states relations" or "women state legislature" with an average length of m=2.9 and a maximum of m=5. One particularity of the TREC queries is that they come with larger description and narrative fields that allow the extraction of larger keyword queries. We indexed the collection with BM25 and a standard TF·IDF scoring model [13].

We imported movie information from the Internet Movie Database IMDB[2] for more than 375,000 movies and more than 1,200,000 persons (actors, directors, etc.) into a four-attribute relational table with the schema `Movies(Title, Genre, Actors, Description)` where `Title` and `Description` are text attributes and `Genre` and `Actors` are set-valued categorical attributes. `Genre` typically contains two or three genres, and actors were limited to those that appeared in at least five different movies. For similarity scores among genres and among actors we precomputed the Dice coefficient for each pair of Genre values and for each pair of actors that appeared together in at least five movies. So the similarity for genres or actors x and y is set to

$$\frac{2 \cdot \#\{\text{movies containing } x \text{ and } y\}}{\#\{\text{movies containing } x\} + \#\{\text{movies containing } y\}}$$

and the index list for x contains entries for similar values y, too, with scores weighted with the similarity of x and y. A typical query is `Title="War" Genre=SciFi Actors="Tom Cruise" Description="alien, earth, destroy"`. We compiled 20 queries of this kind by asking colleagues.

[1]`http://www-nlpir.nist.gov/projects/terabyte/`
[2]`http://www.imdb.org`

The Internet Traffic Archive[3] provides a huge HTTP server log with about 1.3 billion HTTP requests from the 1998 FIFA soccer world championship. We aggregated the information from this log into a relational table with the schema `Log(interval,userid,bytes)`, aggregating the traffic (in bytes) for each user within one-day intervals. Queries ask for the top-k users, i.e., the k users with the highest aggregated traffic, within a subset of all intervals (like "from June 1 to June 10"); our query load consists of 20 such queries.

We compared our new algorithms against the best prior methods. The main competitors turned out to be NRA and CA. We also implemented a *full merge* of the index lists (followed by a partial sort to obtain the top-k results). The full merge is not very competitive in cost, because each element is accessed, but it is actually a tough competitor in terms of running time, because of the significant bookkeeping overhead incurred by all the treshold methods. We further computed the lower bound from Sec. 2.5 to assess how close our algorithms get to the optimum.

We also ran our experiments for the RA-extensive threshold algorithms TA, Upper[5, 22] and Pick[5]. In our setting, where both sorted and random access is possible and a random access is much more expensive than a sorted access (the lowest ratio we consider is 100), all these methods performed considerably worse than even the full merge baseline, in terms of both costs and running times, and for all values of k and c_R/c_S we considered. For example, for k=10 and c_R/c_S=1000 on Terabyte-BM25, they resulted in total cost 72,389,140 (TA), 31,496,440 (Upper), and 3,798,549 (Pick), compared to 2,890,768 for the full merge, 788,511 for NRA and 386,847 for our best method. We therefore did not include these methods in our charts. Note that, as we discussed in Sec. 1.3, MPro, Upper and Pick were actually designed for a different setting, where some lists are accessible by random access only.

We focus on experiments with the Terabyte collection with BM25 scores as this is the most challenging and most realistic collection and scoring model; main results for the two other collections are presented afterwards. Unless stated otherwise, we use a cost ratio c_R/c_S=1,000 and a block size b=32,768 for the experiments which is reasonable for our implementation. We report the average query cost as $COST = \#SA + c_R/c_S \cdot \#RA$ computed over the whole batch of queries as our primary performance measure.

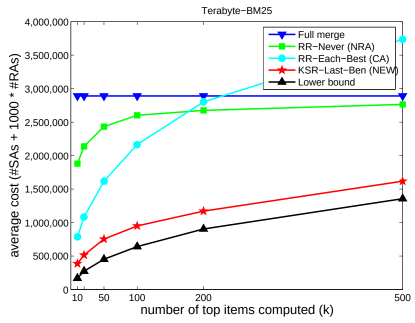

Figure 3: Average costs for Terabyte-BM25 of our best algorithm compared to various baselines and a computed lower bound, for varying k.

[3]`http://ita.ee.lbl.gov`

6.2 Terabyte Runs

Fig. 3 presents the average cost savings of our best approach (`KSR-Last-Ben`) which outperforms all our three baselines by factors of up to 3. Even for k=1,000, there is a 50% improvement over all three baselines. Note that the end user of top-k results (as in web search) would typically set k to 10–100, whereas application classes with automated result post-processing (such as multimedia retrieval) may choose k values between 100 and 1,000. Especially remarkable is the fact that we consistently approach the absolute lower bound by about 20% even for large k, whereas both CA and NRA increasingly degenerate; CA even exceeds the FullMerge baseline in terms of access cost for $k > 200$.

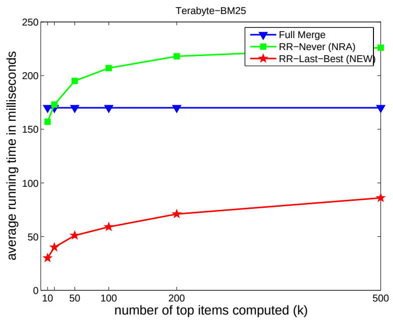

Figure 4: Average running times in milliseconds of our best algorithm compared to FullMerge and NRA, for Terabyte-BM25 and varying k.

Fig. 4 shows the average runtimes we achieve per query, measured on a 2-processor Opteron 250 server with 8 gigabytes of RAM and all data loaded from a SCSI raid. Average runtimes for our algorithms are in the order of 30–60ms for 10≤ k ≤100, even when the total list length is in the millions, which outperforms the NRA and FullMerge baselines by a factor of up to 5. Interestingly, for k >20 our true baseline for measuring runtimes is no longer CA, because it is already outperformed by the DBMS-style FullMerge. Here, NRA is already out of the question because of its high overhead in index access costs (Fig. 3) and its additional need for candidate bookkeeping, whereas the amount of access costs saved by our improved scheduling approaches (`KSR-Last-Ben`) more than compensates the bookkeeping overhead. To pick just one example, the full merge on the query "kyrgyzstan united states relations", which has a total list volume of over 15 million doc ids, takes about one second, while our best top-k algorithms, by scanning only about 2% of this volume and by doing about 300 well-targeted random lookups, process the same query in about 10ms.

6.2.1 *Sorted Access Scheduling*

To analyze the benefit of our Knapsack-driven SA scheduling approaches, we fix the RA scheduling to `Last-Best` and focus on the individual SA scheduling performance of the Knapsack optimizations. Fig. 5 shows relatively low performance gains in between 2–5% for BM25 scores compared to round-robin. For more skewed distrubutions such as TF·IDF, we observe larger benefits of up to 15% for $k \geq 50$. Here, the more sophisticated benefit-optimized Knapsack (`KBA`) wins overall.

6.2.2 *Random Access Scheduling*

We fix the SA scheduling to the basic round-robin strategy and analyze our different RA scheduling approaches.

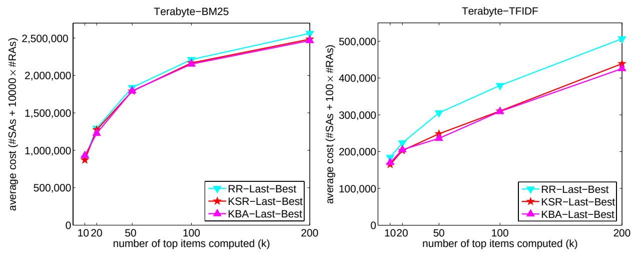

Figure 5: Average cost for the different SA scheduling approaches for Terabyte with a BM25 (left) and a TF·IDF model (right), for varying k.

Fig. 6 shows that we gradually improve our RA scheduling performance as we move from the original CA baseline over the simple `Last-Best` strategy toward the more sophisticated cost-driven scheduling `Last-Ben`. Interestingly, the step from `RR-Each-Best` (CA) to `RR-Last-Best` already provides 90% of the overall gain we can achieve, whereas the more complex `RR-Last-Ben` achieves about 10% more cost savings with an overall factor of about 2.3 compared to the CA baseline.

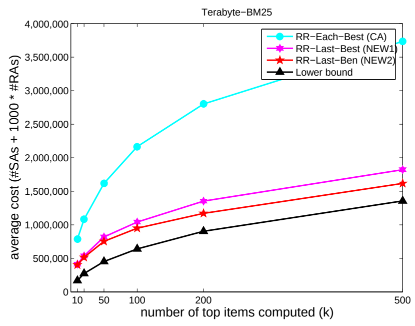

Figure 6: Average cost for the different RA scheduling approaches for Terabyte-BM25, for varying k.

6.2.3 *Varying the Query Size*

In the next setup we increase the query size m for the Terabyte setting by also taking terms from the query descriptions into account, increasing the average query size to m=8.3 with a maximum of m=15 terms, simulating query expansion. Increasing the query dimensionality m yields further performance gains of up to a factor of 2.3 over NRA and a factor of 4 over CA (see Fig. 7). Note that NRA and CA essentially scan the whole lists for the larger m; then NRA has essentially the same costs as FullMerge, while CA costs almost twice as much, due to its proportional number of random accesses.

6.2.4 *Varying the c_R/c_S Ratio*

By tuning the c_R/c_S ratio we can easily simulate different systems setups. Obviously, large ratios punish RA and make NRA or even FullMerge more attractive. This is the case in systems with high sequential throughput and relatively low RA performance (e.g., c_R/c_S=10,000 for mostly raw disk accesses with hardly any caching as opposed to c_R/c_S=100 for a DBMS with lower sequential throughput but higher RA performance through caching). Fig. 8 shows that for low values of c_R/c_S between 100 and 1,000, the combined scheduling strategies provide the highest cost savings with a factor of more than 2 for k=100. Even when only very few

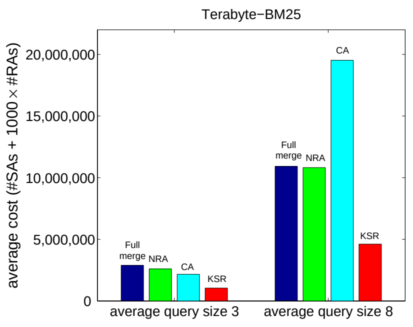

Figure 7: Average costs for Terabyte-BM25 of our best algorithm (`KSR-Last-Ben`) compared to various baselines, for shorter queries (left) and longer queries (right), for $k = 100$.

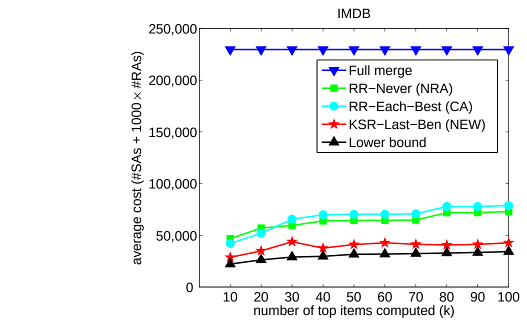

Figure 9: Average cost for IMDB of our best algorithm compared to various baselines and a computed lower bound, for varying k.

RA are allowed, a clever scheduling can still make a decisive difference and improve over NRA or FullMerge.

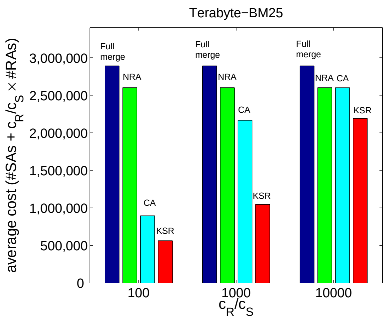

Figure 8: Average cost for Terabyte-BM25 of our best algorithm (`KSR-Last-Ben`) compared to various baselines with $c_R/c_S = 100$ (left), $c_R/c_S = 1,000$ (middle) and $c_R/c_S = 10,000$ (right), for $k = 100$.

6.3 Various Data Collections

6.3.1 IMDB

The largest index lists derived from the IMDB collection with up to a length of 285,000 entries are generated by the categorical attributes such as Genres and Years, whereas the largest inverted lists from text contents only yield a few thousand entries which are typically scanned through by the first block. This makes the collection provide an interesting mixture of short textual lists with quickly decreasing scores and longer lists of categorical values with a low skew and many score ties. Fig. 9 shows that the performance gains here are a bit less than for Terabyte with a factor of 1.5 to 1.8 for $10 \leq k \leq 200$. For this particular combination of lists and mixture of score distributions, all top-k algorithms outperform the FullMerge baseline by a large margin, for wide ranges of k. Note that we are still able to stay very close to the lower bound compared to CA and NRA.

6.3.2 HTTP Worldcup Log

The HTTP Worldcup log yields highly skewed score distributions with a few users having downloaded up to 750 MB per day, whereas the average traffic per user and day lies between 50-100 KB. Fig. 10 shows that CA (which is already close to optimal) becomes more competitive to our best algorithm (`KBA-Last-Ben` here) with only a factor of about 1.2 additional cost for k up to 100, because a few random accesses on the currently best-scored items typically suffice to yield the final top-k results. `KBA-Last-Ben` almost touches the lower bound for wide ranges of k. Note that for these skewed distributions, the benefit-optimized Knapsack `KBA` yields the better basis for SA scheduling. Also note that here NRA ends up scanning the full lists already for relatively small k.

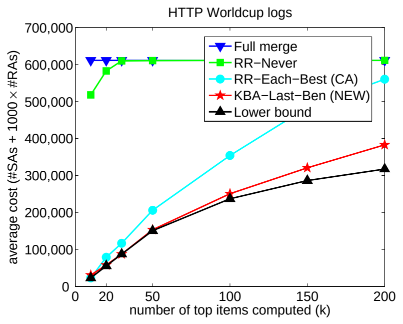

Figure 10: Average cost for the HTTP Worldcup logs of our best algorithm compared to various baselines and a computed lower bound, for varying k.

6.4 Discussion of Experiments

For many real-word data sets and score distributions, Fagin's originally proposed CA algorithm already yields a tough baseline. Except for extremely skewed distributions and small values of k, NRA is out of the question, because there is typically only a marginal difference between the final scores of the kth and $(k + 1)$-ranked result which makes the best- and worstscores converge very slowly and leads to a very late threshold termination (Fig. 3). On the other extreme, TA with its high overhead in random I/O is a viable choice only for setups with an extremely low c_R/c_S ratio. Our experiments demonstrate that our proposed algorithms perform much better than CA which is considered the most versatile variant of Fagin's algorithm, especially for larger k.

A comparison with two artificially generated *Uniform* and *Zipf* distributions for Terabyte reveals that for uniformly distributed scores, the round-robin SA scheduling already provides the best approach, whereas for more skewed distributions (e.g., TF·IDF, Fig. 5, or Zipf) the Knapsack-based optimizations take effect. Fortunately, the Knapsack implementations tend to converge exactly to such a round-robin-like SA schedule in the Uniform case, hence they do not degenerate, but also cannot improve much over the round-robin baseline in this case. Generally, a few judiciously scheduled RA have the potential to yield an order of magnitude higher cost savings than the best SA scheduling could.

For all setups, our algorithms that postpone random accesses to a late, more cost-beneficial phase and hence gather more information about the intermediate top-k and candidate items outperform their algorithmic pendants that eagerly trigger random accesses after each batch of sorted accesses (Fig. 6). For all values of k and cost ratios c_R/c_S, our probabilistic extensions outperform the baseline algorithms by a large margin; moreover, they never degenerate or lead to higher access costs than their non-probabilistic counterparts. The simple Last-Probing approach with its heuristic stopping criterion is already a very solid basis; the cost-based Ben-Probing beats it merely by another 10% of costs saved and in fact comes close to the lower bound for many queries and collections (Fig. 3, 9, and 10). Note that the iterative evaluation of the cost formulas in Sec. 4, 5.1, and 5.2 is fairly light-weight so that the overhead of running the cost models for all candidates after a batch of b SAs is acceptable with regard to the costs saved in physical I/O.

7. CONCLUSIONS

This paper presents a comprehensive algorithmic framework and extensive experimentation for various data collections and system setups to address the problem of index access scheduling in top-k query processing. Unlike more aggressive pruning strategies proposed in the literature [19, 24, 29] that provide approximate top-k results, the methods we presented here are non-approximative and achieve major runtime gains of factors up to 5 over existing state-of-the-art approaches with no loss in result precision. Moreover, we show that already the simpler methods of our framework, coined the `Last` strategies, provide the largest contribution to this improvement, and the probabilistic extensions get very close to a lower bound for the optimum cost. Future work could investigate the combination of our approach with approximative pruning strategies; also the extension of our index access scheduling framework for processing XML data along the lines of [18, 23] would be very interesting.

8. REFERENCES

[1] S. Agrawal et al. Automated ranking of database query results. *CIDR*, 2003.

[2] R. Avnur, J. M. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD 2000*, pages 261–272, 2000.

[3] W.-T. Balke, U. Güntzer, J. X. Zheng. Efficient distributed skylining for web information systems. *EDBT 2004*, pages 256–273, 2004.

[4] H. Bast et al. IO-Top-k: Index-access optimized top-k query processing. Techn. Rep. MPI-I-2006-5-002, MPI Informatik, 2006. `http://domino.mpi-sb.mpg.de/internet/reports.nsf/NumberView/2006-5-002/`.

[5] N. Bruno, L. Gravano, A. Marian. Evaluating top-k queries over web-accessible databases. *ICDE 2002*, pages 369–380, 2002.

[6] C. Buckley, G. Salton, J. Allan. The effect of adding relevance information in a relevance feedback environment. *SIGIR 1994*, pages 292–300, 1994.

[7] P. Cao, Z. Wang. Efficient top-k query calculation in distributed networks. *PODC 2004*, pages 206–215, 2004.

[8] M. J. Carey, D. Kossmann. On saying "enough already!" in SQL. *SIGMOD 1997*, pages 219–230, 1997.

[9] K. C.-C. Chang, S.-W. Hwang. Minimal probing: supporting expensive predicates for top-k queries. *SIGMOD 2002*, pages 346–357, 2002.

[10] S. Chaudhuri, L. Gravano, A. Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE TKDE*, **16**(8):992–1009, 2004.

[11] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, **31**(2):109–118, 2002.

[12] R. Fagin, A. Lotem, M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, **66**(4):614–656, 2003.

[13] D. A. Grossmann, O. Frieder. *Information Retrieval.* Springer, 2005.

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

[15] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, **13**(3):207–221, 2004.

[16] I. F. Ilyas et al. Rank-aware query optimization. *SIGMOD 2004*, pages 203–214, 2004.

[17] Y. E. Ioannidis. The history of histograms (abridged). *VLDB 2003*, pages 19–30, 2003.

[18] R. Kaushik et al. On the integration of structure indexes and inverted lists. *SIGMOD 2004*, pages 779–790, 2004.

[19] N. Lester et al. Space-limited ranked query evaluation using adaptive pruning. *WISE 2005*, pages 470–477, 2005.

[20] C. Li et al. RankSQL: Query algebra and optimization for relational top-k queries. *SIGMOD 2005*, pages 131–142, 2005.

[21] X. Long, T. Suel. Optimized query execution in large search engines with global page ordering. *VLDB 2003*, pages 129–140, 2003.

[22] A. Marian, N. Bruno, L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM TODS*, **29**(2):319–362, 2004.

[23] A. Marian et al. Adaptive processing of top-k queries in XML. *ICDE 2005*, pages 162–173, 2005.

[24] A. Moffat,J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM TOIS*, **14**(4):349–379, 1996.

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

[26] M. Persin, J. Zobel, R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *JASIS*, **47**(10):749–764, 1996.

[27] W. H. Press et al. *Numerical Recipes in C.* Cambridge University Press, 1992.

[28] M. Theobald, R. Schenkel, G. Weikum. An efficient and versatile query engine for TopX search. *VLDB 2005*, pages 625–636, 2005.

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

[30] H. Yu et al. Efficient processing of distributed top-k queries. *DEXA 2005*, pages 65–74, 2005.