

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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12-15, 2006, Seoul, Korea.

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

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)	Round 2 (SA on 1,2,3)	
Doc17 : [0.8 , 2.4]	Doc17 : [1.5 , 2.0]	
Doc25 : [0.7 , 2.4]	Doc25 : [0.7 , 1.6]	
Doc83 : [0.9 , 2.4]	Doc83 : [0.9 , 1.6]	
unseen: ≤ 2.4	unseen: ≤ 1.4	
Round 3 (SA on 2,2,3!)	Round 4 (RA for Doc17)	
Doc17 : [1.5 , 2.0]	Doc17 : 1.7	
Doc83 : [1.4 , 1.6]	all others < 1.7	
unseen: ≤ 1.0	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].

