Anytime Ranking for Impact-Ordered Indexes

Jimmy Lin[1] and Andrew Trotman[2]

[1] David R. Cheriton School of Computer Science
University of Waterloo

[2] eBay Inc.

ABSTRACT

The ability for a ranking function to control its own execution time is useful for managing load, reigning in outliers, and adapting to different types of queries. We propose a simple yet effective anytime algorithm for impact-ordered indexes that builds on a score-at-a-time query evaluation strategy. In our approach, postings segments are processed in decreasing order of their impact scores, and the algorithm early terminates when a specified number of postings have been processed. With a simple linear model and a few training topics, we can determine this threshold given a time budget in milliseconds. Experiments on two web test collections show that our approach can accurately control query evaluation latency and that aggressive limits on execution time lead to minimal decreases in effectiveness.

Categories and Subject Descriptors: H.3.4 [Information Storage and Retrieval]: Systems and Software

Keywords: score-at-a-time query evaluation; impact scores

1. INTRODUCTION

Anytime algorithms are algorithms where the quality of results improves as the computation time increases [21]. Typically, such algorithms return a valid solution even if interrupted before they naturally complete. This idea was introduced in the mid-1980s by Dean and Boddy [9] in the context of time-dependent planning. Applied to information retrieval, an anytime ranking function is able to provide a document ranking in response to a user's query, given an arbitrary time constraint. We would, of course, expect the output quality to rise as the time budget increases.

This idea is relevant to search because managing query latency is an important aspect of modern information retrieval, particularly in a web search context. Users are impatient and latency has measurable costs: for example, Brutlag [5] reports for Google search that artificially injecting delays ranging 100 to 400 ms reduces the daily number of searches per user by 0.2% to 0.6%. Query latencies are typically managed by partitioning the document collection across many (in the case of the web, thousands) of servers such that the latency at each individual server is small. However, this is often not enough. We see at least three compelling applications for anytime ranking functions:

First, they can be applied for load shedding. During periods of unexpectedly large query loads (e.g., flash mobs) it would make sense to restrict the running time of the ranking algorithm. Although this may come at some cost in effectiveness, degrading quality slightly for *everyone* is often preferable to long latencies (or even timeouts) for *some*.

Second, they can be applied to control variance in execution times, particularly outliers called "tail latencies" [8]. Partitioned systems are particularly vulnerable to this phenomenon, since overall latency is dictated by the latency of the slowest component. An anytime ranking function can be configured to "reign in" these tail latencies without affecting the majority of the queries.

Third, anytime ranking functions can be used in conjunction with effectiveness prediction techniques [11] to treat "easy" and "hard" queries differently (e.g., spend less time on easy queries). In a multi-stage retrieval architecture where the initial ranking serves as input to machine-learned ranking models [4], this might yield the same level of effectiveness with less overall computational effort.

We present a novel anytime ranking algorithm on impact-ordered indexes in main memory. Building on a score-at-a-time query evaluation strategy, which processes postings segments in decreasing order of impact scores, we add an early termination check to stop when a specified number of postings ρ has been processed. A simple linear model with a few training topics allows us to work backwards from a time budget (in milliseconds) to the proper setting of ρ. Experiments on two web test collections show that our approach is both simple and effective: we can accurately control query evaluation latency and we find that aggressive limits on execution time leads to minimal decreases in effectiveness.

2. BACKGROUND AND RELATED WORK

Following the standard formulation of ranked retrieval, we assume that the score of a document d with respect to a query q can be computed as an inner product: $S_{d,q} = \sum_{t \in d \cap q} w_{d,t} \cdot w_{q,t}$, where $w_{d,t}$ is the weight of term t in document t and $w_{q,t}$ represents the weight of term t in the query. The goal of top k retrieval is to return the top k documents ordered by S. Typically, w's are a function of term frequency, document frequency, and the like. This formula-

ICTIR'15, September 27–30, Northampton, MA, USA.

ACM 978-1-4503-3833-2/15/09 ...$15.00.
DOI: http://dx.doi.org/10.1145/2808194.2809477.

tion captures traditional vector-space models, probabilistic models such as BM25, as well as language modeling and divergence from randomness approaches.

Nearly all modern search engines depend on an inverted index for top k retrieval. As is common today, we assume that the entire index and all associated data structures reside in main memory. The literature describes a few ways in which inverted indexes can be organized: In *document-ordered* indexes, postings lists are sorted by document ids in increasing order. Term frequencies are stored separately. In *frequency-ordered* indexes, document ids are grouped by their term frequencies; within each grouping, document ids are sorted in increasing order, but the groupings are arranged in decreasing order of term frequency. In *impact-ordered* indexes, the focus of this work, the actual score contributions of each term (i.e., the $w_{d,t}$'s) are pre-computed and quantized into what are known as *impact scores*. We refer to a block of document ids that share the same impact score as a postings segment. Within each segment, document ids are arranged in increasing order, but the segments themselves are arranged by decreasing impact score. Regardless of the index organization, the postings are usually compressed with integer coding techniques. There has been plenty of work on index compression, which is beyond the scope of this work, but see a recent study for details [17].

Different query evaluation techniques exhibit affinities for different index organizations. Document-at-a-time (DaaT) techniques, which are the most popular today, work well with document-ordered indexes and term-at-a-time (TaaT) techniques work well with frequency-ordered indexes. Similarly, score-at-a-time (SaaT) strategies take advantage of impact-ordered indexes. A review of query evaluation techniques is beyond the scope of this short paper, but we refer readers to a survey by Zobel and Moffat [22].

The idea of anytime ranking is not entirely new. In the context of learning to *efficiently* rank, Wang et al. [20, 19] proposed machine-learned ranking models that can control their own execution costs. Along the same lines, Cambazoglu et al. [6] introduced early-exit optimizations for ensembles of machine-learned rankers. In a DaaT query evaluation strategy, Macdonald et al. [14] incorporate query performance prediction to facilitate query scheduling. Although Zobel and Moffat [22] allude in passing to an approach along the lines of what we propose, we are not aware of any work that has detailed a concrete implementation with appropriate performance evaluations.

3. ANYTIME RANKING

We take as a starting point the standard inner-product formulation of ranked retrieval described above. In impact-ordered indexes, the $w_{d,t}$'s are pre-computed and quantized into b bits (called its impact score). Here, we use BM25 term weighting. The literature discusses a number of techniques for quantizing the term weights, but in this work we adopt the *uniform* quantization method of Anh et al. [1]:

$$i_{d,t} = \left\lfloor \frac{w_{d,t} - \min(w_{d,t})}{\max(w_{d,t}) - \min(w_{d,t})} \times 2^b \right\rfloor \quad (1)$$

which is an index-wide linear scaling of the term weights and b is the number of bits used to store the impact. In our implementation we set $b = 8$. Crane et al. [7] showed that this setting achieves effectiveness that is indistinguishable from using exact term weights.

3.1 Index Organization

Our indexes are organized as follows: The dictionary provides the entry point to each postings list; each term points to a list of tuples containing (score, start, end, count). Each tuple, which we refer to as a header, corresponds to a postings segment with a particular impact score; start and end are pointers to the beginning and end of the segment data, and count stores the number of documents in that segment. Segments for each term are ordered in decreasing impact score and within each segment, documents are ordered by increasing document id.

Document ids are compressed with QMX [17], which can be thought of as an extension of the Simple family [3] that takes advantage of SSE (Streaming SIMD Extensions) instructions in the x86 architecture. Experiments [17] have shown QMX to be more efficient to decode than SIMD-BP128 [13] (previously the most decoding efficient overall) and competitive with all SIMD and non-SIMD techniques in terms of size.

Following convention, postings code differences between document ids (called d-gaps) instead of the document ids directly. In our case, we compute gaps with respect to the document id four positions earlier, i.e., the fifth integer encodes the difference relative to the first, the sixth relative to the second, etc. This approach takes advantage of a SIMD instruction to decode four gaps in one instruction.

3.2 Query Evaluation

Our anytime ranking algorithm builds on a score-at-a-time (SaaT) query evaluation strategy. The algorithm begins by fetching the headers of all postings lists that correspond to the query terms and sorting the headers by decreasing impact score. The postings segments are then processed in this order. For each document id in a segment, the impact score is added to the accumulator, and thus the final result is an unsorted list of accumulators. To avoid sorting this list, a heap of the top k can be maintained during processing. That is, after adding the current impact score to the accumulator, we check to see if its score is greater than the smallest score in the heap; if so, the pointer to the accumulator is added to the heap. The heap keeps at most k elements, and we break ties arbitrarily based on document id.

With SaaT query evaluation, several approaches to accumulator management have been proposed [15, 1, 2, 12]. In this work, we implement the approach of Jia et al. [12]. Since the impact scores are 8 bits, and queries are generally short, it suffices to allocate an array of 16-bit integers, one per document indexed by the document id; modern hardware has ample memory to keep the accumulators in memory. This approach is much simpler than other accumulator management strategies focused on accumulator pruning (e.g., [15, 2]), which made sense when memory was scarce.

Since we are processing postings segments in decreasing impact order, we can terminate at any time. By definition, the segments are processed in decreasing importance: larger score contributions will be added earlier such that the ranking is gradually refined as query evaluation progresses. Early termination to satisfy a time budget is controlled by a parameter ρ, the maximum number of postings to process. Translating a time budget (in milliseconds) into ρ is accomplished using a simple linear model, described later.

The query evaluation algorithm keeps a cumulative count of the number of postings it has processed. Before process-

Name	# Docs	TREC Topics
ClueWeb09b	50,220,423	51-200 ('10–'12)
ClueWeb12-B13	52,343,021	201-300 ('13–'14)

Table 1: Summary of TREC collections and topics used in our experiments.

ing the next postings segment, it checks if processing this particular segment will exceed ρ; if so, we break out of the processing loop. At this point, all that remains is to extract the top k results from the heap (which has a constant cost). This means that our algorithm is also *interruptible*, in that the algorithm does not need to know in advance the time budget; we can demand the termination of the algorithm at any time with an external signal.

One might wonder why it is necessary to build a query efficiency prediction model: why not simply check the query latency after processing each segment? This would not be feasible because the system calls for time measurement are costly operations (relative to processing postings). Even if one wanted to use such measurements to more carefully control query evaluation latency, it still makes sense to use a prediction model to guide the timing of the system calls.

4. EXPERIMENTAL SETUP

Our anytime ranking algorithm is implemented in C++ (compiled with gcc version 4.9.1) and part of an open-source retrieval engine called JASS.[1] Instead of implementing a complete search engine, we use inverted indexes built by the ATIRE system [18],[2] which saved us from having to write a separate indexer. Our system reads indexes generated by ATIRE and rewrites data into an internal format.

Experiments used two standard TREC web test collections: ClueWeb09 (category B), CW09 for short, and ClueWeb12-B13 (i.e., "category B"), CW12 for short. Details for these collections are provided in Table 1, showing the sizes of each and the corresponding topics used to evaluate effectiveness. For simplicity, we kept collection processing to a minimum: for each document, all invalid UTF-8 characters were converted into spaces, alphabetical characters were separated from numerical characters; stemming was applied but no additional document cleaning was performed other than markup tag removal. All experiments were conducted on a server with dual Intel Xeon E5-2680 v3 2.5GHz (12 cores) with 768 GB RAM, running Red Hat Enterprise Linux (RHEL) 6. All experiments were conducted on a single thread on an otherwise idle machine.

For each of the two test collections, the first ten topics were used for training and the remaining topics were used for testing. Our efficiency metric is query latency, the time it takes for our query evaluation engine to produce the top k ranking, measured with the `chrono` library. Measurements exclude file I/O costs, i.e., we keep track of the time it takes to materialize the top k documents in main memory, but do not include the time taken to write the output files for evaluation. We also exclude one-time startup costs such as loading dictionaries, postings, etc. into main memory. We used NDCG@10 as the effectiveness metric, and thus our experiments retrieved only the top 10 results.

[1] `https://github.com/lintool/JASS`
[2] `http://atire.org/`

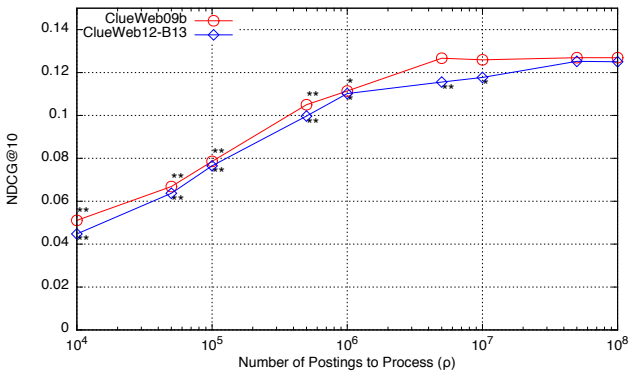

Figure 1: Effectiveness for different ρ settings.

5. RESULTS

Our first set of experiments was designed to highlight the relationship between ρ (number of postings to process) and effectiveness. We accomplished this by a parameter sweep across a wide range of ρ values (10k, 50k . . . , 100m) and measuring the effect on NDCG@10; these results, across all topics in both collections, are shown in Figure 1. Significance testing with respect to exhaustive evaluation (of all postings) was conducted using Fisher's two-sided, paired randomization test [16]; following convention, * denotes $p < 0.05$ and ** denotes $p < 0.01$. The rightmost setting of ρ yields identical results to exhaustive evaluation. Note that we do not correct for repeated hypothesis testing, so our tests are conservative. These results show that we can reduce the number of postings processed quite a bit without significantly hurting effectiveness. From this figure, we suggest that setting ρ to 10% of the collection size achieves a reasonable balance between effectiveness and efficiency.

The next step was to train a model for predicting ρ (number of postings to process) given a time budget (in milliseconds). We used the first ten topics of each test collection for training and conducted the same parameter sweep as above, recording per-query latency and the number of postings processed; this procedure was repeated for three trials. Given the 10% heuristic above, we retained only data points where the number of postings processed was less than 10% of the collection, since this is the operating region we wish to focus on. For both CW09 and CW12, the data fit a linear regression well, which corresponds to a model that includes a constant overhead plus a cost per posting processed. For space considerations, we are unable to include the scatter plot of the performance model, but the best fit line for CW09 has a slope of 2×10^{-5} with an intercept of 11.741 (R^2 of 0.944); the best fit line for CW12 has a slope of 3×10^{-5} with an intercept of 18.404 (R^2 of 0.982). In our final model of ρ, we rounded the intercept values up to 12 and 19 for CW09 and CW12, respectively.

Finally, we evaluated our anytime algorithm with time budgets of {25, 50, 100, 150, 200} milliseconds. For each time budget, we used the linear model above to determine the appropriate setting of ρ. Results are shown in Table 2, averaged over three trials. Each row shows a particular time budget; relative effectiveness differences are computed with respect to exhaustive processing (final row denoted "max"). Note that NDCG@10 is computed over *all* topics to facilitate comparison with published results, but the remaining columns are with respect to the *test* topics only (140 for

target	ClueWeb09b						ClueWeb12-B13					
	NDCG@10	time	ET	miss	mean	max	NDCG@10	time	ET	miss	mean	max
25ms	0.1076 (−15%) **	22.3	112	45	0.79	2.3	0.0798 (−36%) **	25.4	87	76	0.64	1.2
50ms	0.1159 (−8.7%) *	37.2	97	3	0.77	1.5	0.1102 (−12%) *	47.5	74	49	1.3	3.2
100ms	0.1244 (−2.0%)	59.5	77	0	-	-	0.1118 (−11%) **	80.1	63	9	1.7	3.8
150ms	0.1280 (+0.9%)	75.6	71	0	-	-	0.1172 (−6.2%) *	104	55	0	-	-
200ms	0.1264 (−0.4%)	87.8	62	0	-	-	0.1149 (−8.1%) **	122	49	0	-	-
max	0.1269	160	0	-	-	-	0.1250	291	0	-	-	-

Table 2: Evaluation of our anytime algorithm for various time budgets and exhaustive processing ("max"). Times are measured in milliseconds; NDCG@10 is computed over all topics, but other columns are with respect to the test topics only (see text for explanation).

CW09 and 90 for CW12): The column "time" shows the mean latency for that condition. The column "ET" shows the number of topics that terminated early. The column "miss" shows the number of topics that missed the time budget on average across the three trials. For these topics, "mean" shows the average deficit (i.e., how far past the allotted time) in milliseconds, and "max" is the maximum. For example, in the 25ms condition, 45 out of 140 queries in CW09 exceeded the time budget, by an average of 0.79ms and a maximum of 2.3ms. For reference, exhaustively processing all postings averaged 160ms for CW09 (max 1.51s); 291ms for CW12 (max 1.21ms).

Overall, we are able to quite precisely control the execution time of our anytime ranking function. In the cases where the time budget is violated, the delays are minor, and it would be simple to add a constant "safety factor" if we desired a more stringent observance of the time budget. Our algorithm becomes more conservative as the time budget increases, since segments with lower impact scores tend to be longer and we never partially process a postings segment. Note that exhaustively processing all postings takes much longer on CW12 than CW09, which means that the same time budget leads to more effectiveness compromises; thus, we see significant losses in NDCG@10 for CW12.

Finally, these experiments highlight the ability of our anytime algorithm to control tail latencies. For example, consider the 200ms case with CW09: the mean latency across all topics is only 87.8ms and $140 - 62 = 78$ topics (the "ET" column) in fact finish processing all postings within the time budget. The ρ cutoff in effect aborts queries that are taking too long, without significantly compromising effectiveness.

6. CONCLUSION

Our SaaT strategy represents a very different approach to query evaluation than DaaT algorithms that are popular today [10]. Although we see a few different ways that a DaaT strategy can be modified into an anytime algorithm, they all seem much more complex than our SaaT approach. However, the interesting question is: for a given time budget, can we achieve higher effectiveness with the SaaT approach here or a yet-to-be-developed DaaT anytime algorithm? This remains an open question deserving future investigation.

7. ACKNOWLEDGMENTS

This work was supported by NSF awards IIS-1218043 and CNS-1405688 while the first author was at the University of Maryland. Any opinions, findings, conclusions, or recommendations expressed are the authors' and do not necessarily reflect the views of the sponsor.

8. REFERENCES

[1] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. *SIGIR*, 2001.

[2] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. *SIGIR*, 2006.

[3] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Software: Practice and Experience*, 40(2):131–147, 2010.

[4] N. Asadi and J. Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. *SIGIR*, 2013.

[5] J. Brutlag. Speed matters for Google web search. Technical report, Google, 2009.

[6] B. B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt. Early exit optimizations for additive machine learned ranking systems. *WSDM*, 2010.

[7] M. Crane, A. Trotman, and R. O'Keefe. Maintaining discriminatory power in quantized indexes. *CIKM*, 2013.

[8] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2):74–80, 2013.

[9] T. Dean and M. Boddy. Time-dependent planning. *AAAI*, 1988.

[10] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. *SIGIR*, 2011.

[11] C. Hauff, V. Murdock, and R. Baeza-Yates. Improved query difficulty prediction for the web. *CIKM*, 2008.

[12] X.-F. Jia, A. Trotman, and R. O'Keefe. Efficient accumulator initialisation. *ADCS*, 2010.

[13] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[14] C. Macdonald, N. Tonellotto, and I. Ounis. Learning to predict response times for online query scheduling. *SIGIR*, 2012.

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

[16] M. D. Smucker, J. Allan, and B. Carterette. A comparison of statistical significance tests for information retrieval evaluation. *CIKM*, 2007.

[17] A. Trotman. Compression, SIMD, and postings lists. *ADCS*, 2014.

[18] A. Trotman, X.-F. Jia, and M. Crane. Towards an efficient and effective search engine. *Workshop on Open Source Information Retrieval*, 2012.

[19] L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. *SIGIR*, 2011.

[20] L. Wang, D. Metzler, and J. Lin. Ranking under temporal constraints. *CIKM*, 2010.

[21] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.

[22] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(6):1–56, 2006.