Fast Candidate Generation for Two-Phase Document Ranking: Postings List Intersection with Bloom Filters

Nima Asadi[1,2], Jimmy Lin[3,2,1]
[1]Dept. of Computer Science, [2]Institute for Advanced Computer Studies, [3]The iSchool
University of Maryland, College Park
nima@cs.umd.edu, jimmylin@umd.edu

ABSTRACT

Most modern web search engines employ a two-phase ranking strategy: a candidate list of documents is generated using a "cheap" but low-quality scoring function, which is then reranked by an "expensive" but high-quality method (usually machine-learned). This paper focuses on the problem of candidate generation for conjunctive query processing in this context. We describe and evaluate a fast, *approximate* postings list intersection algorithms based on Bloom filters. Due to the power of modern learning-to-rank techniques and emphasis on early precision, significant speedups can be achieved without loss of end-to-end retrieval effectiveness. Explorations reveal a rich design space where effectiveness and efficiency can be balanced in response to specific hardware configurations and application scenarios.

Categories and Subject Descriptors: H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval

General Terms: Algorithms, Experimentation

Keywords: scalability, efficiency, conjunctive queries

1. INTRODUCTION

There is general consensus that the challenge of document ranking is best tackled using machine learning techniques [11, 9]. This "learning to rank" approach generally assumes that a candidate list of potentially-relevant documents has already been gathered by other means. Thus, learning to rank is actually a *reranking* problem, using, for example, boosted regression trees [5, 9]. Hence, modern web search can be viewed as a two-phase process: candidate generation followed by reranking.

This paper focuses on the problem of candidate generation for learning to rank algorithms. Following previous work, we operationalize this as postings list intersection [4, 17]. Our intuition is that modern rankers are relatively insensitive to the quality of the candidate list. As long as this intermediate product is of "reasonable" quality, the end result will remain good—particularly in the web context due to its emphasis on early precision. Thus, our goal is to generate candidate documents as quickly as possible, even at the cost of introducing approximations.

We see this work as having two main contributions: First, we propose a novel algorithm for postings list intersection that takes advantage of Bloom filters. Evaluations on real world data show that it is fast, yet does not sacrifice end-to-end retrieval effectiveness. Second, we empirically characterize the tradeoff space between effectiveness (result quality), time (retrieval speed), and space (index size), illustrating how these three aspects can be traded off for each other.

2. BACKGROUND AND RELATED WORK

For modern web retrieval, query evaluation is often broken into two phases [4, 17]. In the first phase, a fast, "cheap" algorithm generates a candidate list of potentially-relevant documents (e.g., using interpolated BM25 and static prior). In most cases, queries are processed conjunctively, i.e., only documents that contain *all* the query terms are considered. For web-scale collections, this leads to higher early precision and faster query evaluation [4]. The candidate documents from the first phase are then reranked by a slower, "expensive" but better (machine learned) algorithm (e.g., boosted regression trees [5, 9]). Within this general setup, our work focuses on the candidate generation process.

Conjunctive query processing requires solving the problem of postings list intersection, which has been studied in detail [8, 1, 18]. In particular, we use the eliminator-based "small adaptive" algorithm proposed by Demaine et al. [8] as the baseline. Although Bloom filters have been used in P2P retrieval systems [12] and retrieval based on bit signatures [16], to our knowledge this application is novel.

3. APPROACH

The starting point of this work is that modern learning-to-rank approaches for web search are sufficiently powerful to return high-quality results as long as they are presented with a "reasonable" candidate list of documents—especially based on metrics that focus on early precision. As a result, we can leverage probabilistic data structures to speed up candidate generation (postings list intersection). Given a query Q, the task is to retrieve the top n documents, based on a query-independent score, that contain all the query terms. As is the case with commercial web search engines, we assume that all index structures are held in memory.

In most collections, documents are arbitrarily numbered, so document-sorted indexes do not provide any scoring or ranking. To address this issue, we renumber the collection so that document ids are assigned based on a query-independent score (e.g., PageRank or spam scores). In our case, the smallest document id is assigned to the document estimated to have the highest quality, and documents are successively numbered in order of decreasing page quality, with ties broken arbitrarily. With this "renumbering trick," postings now implicitly capture ranking information, which provides early termination: if we wish to generate top n documents (in terms of the query-independent score), we traverse postings in order and stop when we've gathered enough documents. Finally, since document ids are guaranteed to be in ascending order, we can continue to use efficient gap-based techniques to compress postings. Following standard practice, we use PForDelta [19]. Note that since we focus on postings list intersection, there is no need to store *tf*'s and positional information in the index.

The core contribution of this work is a novel algorithm for postings list intersection using Bloom filters. In our approach, each postings list is stored both as a compressed sequence of integers *and* as a Bloom filter [2]. A Bloom filter is a fast, compact data structure that supports $O(1)$ approximate set membership tests; that is, the Bloom filter representation of a postings list allows us to quickly answer the question, "is this document id contained in the postings list?" The relevant parameters for a Bloom filter are r (bits per element) and k (number of hash functions), which we fix for all Bloom filters in the index. Given a parameter setting, we can analytically model the false positive rate, as provided by Bose et al. [3]. In the interest of space, we do not repeat the bounds or the derivation here.

The postings list intersection algorithm proceeds as follows: for a query Q with $|Q|$ terms, we find the term q with the smallest document frequency (i.e., least frequent query term) and look up its standard postings list. We refer to this as the *base* postings list. This postings list is then traversed: walking down the list of document ids, the algorithm probes the Bloom filter representation of postings lists corresponding to the other query terms to compute the set intersection. A document is added to the candidate list if all membership tests pass. Since the base postings list is sorted by the query-independent score, to generate a list of n candidate documents we only need to traverse the base postings list until the n_{th} matching document is decoded, thereby allowing early termination. In our experiments, we set n to be 10000, but this parameter can be tuned to the application scenario. The approximation aspect of our algorithm lies in the fact that Bloom filters can produce false positives—that is, a filter can assert that an element is contained within it, even when in reality the element was never inserted.

When constructing Bloom filter representations for a collection of N documents, it is clear that by fixing r (bits per element), the size of the Bloom filters for postings lists that have more than $\theta_I = \frac{N}{r}$ elements exceeds N bits. Thus, for very frequent terms, we replace the Bloom filter with a bit-array index. Not only does this reduce the false positive rate to zero, but it also obviates the need to compute hash values, thereby increasing the speed of the probes.

For postings lists with $n < \theta_I$ elements, we build a Bloom filter with k hash functions and $r \times n$ bit positions. We use the Jenkins integer hash function, with the form $h(x, S)$, where S is a seed. For Bloom filter setting $k = 1$, we simply use a large prime number as the seed and compute $h(x, S)$ mod $(n \times r)$ as the hash value, for a particular setting of r on a postings list with n postings. For $k > 1$, the n_{th} hash value is computed by seeding the $(n-1)_{th}$ hash value to $h(x, S)$ mod $(n \times r)$. Recognizing the accuracy, time, and space tradeoffs discussed above, we experimented with a range of settings to empirically quantify the effects.

4. EXPERIMENTAL SETUP

We performed experiments on the first English segment of the ClueWeb09 collection. For our query-independent score, we used the Waterloo spam scores [7]. After document id reassignment (see Section 3), we compressed postings lists (document ids only) using PForDelta with a block size of 128. The compressed postings lists total 13.88 GB in size. We used two different sets of queries for evaluation. The 50 queries from the TREC 2009 web track were used for effectiveness experiments. For efficiency, we used the AOL query log [15], which contains around 10 million queries.

Implementations of the algorithms in this paper are in Java, built on top of the open-source Ivory toolkit [13]. We used LinkedIn's PForDelta implementation in the open-source Kamikaze package.[1] Following standard practice, we used gap-compressed PForDelta [19] (block size of 128) for the baseline implementation (small adaptive). The choice of Java puts us at a disadvantage compared to, say, implementations in C/C++. Since our task is an intermediate step in a retrieval pipeline, ease of component integration is important. For this, Java holds a number of advantages in today's software ecosystem—for example, Twitter's real-time search engine, which serves over 2 billion queries per day, is implemented in Java [6]. Regardless, since all implementations are in the same language, the comparison remains fair—were we to reimplement everything in C/C++, the relative results should remain the same. We have put in a best faith effort to optimize the small adaptive algorithm in our implementation. Thus, we are confident that observed differences are not caused by neglect or an underperforming baseline. Finally, we note that all implementations are presently single-threaded.

Although the focus of this work is on fast postings list intersection, to illustrate end-to-end retrieval effectiveness, we implemented a simple learning-to-rank algorithm to rerank the candidate documents. We used the simple greedy feature selection algorithm proposed by Metzler [14] with a standard set of features, which include basic information retrieval scores (e.g., BM25 and language modeling scores), term proximity features (e.g., exact phrase, ordered and unordered windows), and query-independent features (e.g., spam score). There are a total of 43 features. We performed two-fold cross-validation optimizing NDCG.

Experiments were performed on a server running Red Hat Linux, with dual Intel Xeon "Westmere" quad-core processors (E5620 2.4GHz) and 128GB RAM. This particular architecture has a 64KB L1 cache per core, split between data and instructions; a 256KB L2 cache per core; and a 12MB L3 cache shared by all cores (of a single processor). As stated previously, we assume all index structures are completely held in memory. This is not an unreasonable assumption given the capabilities of commodity servers today—and as we shall see, the memory requirements are modest.

There are three important considerations in the design of search engines: effectiveness (result quality), time (query evaluation speed), and space (index size). The last two are straightforward to measure. Query evaluation speed is measured in terms of latency, the per-query time required for

[1] `http://sna-projects.com/kamikaze/`

$r \backslash k$	1	2	3
8	74.95	85.34	89.09
16	84.80	93.63	97.59
24	93.96	98.04	99.81

Table 1: Relative recall for the TREC 2009 queries compared to exact postings list intersection.

performing postings list intersection. Index size can be easily computed. A setting of r yields Bloom filters of a particular size (unaffected by k). We apply the optimization described in Section 3, where very long postings lists are replaced with bit arrays, such that the maximum size required for any term is the size of the document collection in bits.

Finally, effectiveness: we report both component-level and end-to-end metrics. At the component level, relative recall with respect to exact postings list intersection (i.e., small adaptive) best captures output quality. That is, of all relevant documents retrieved by the *exact* algorithm, what fraction is returned by our *approximate* algorithm (factoring in errors introduced by the Bloom filters). More precisely, relative recall is computed via micro-averaging (i.e., computed per topic, then averaged across topics); this has the effect of disproportionately weighting topics that have fewer relevant documents (which is desirable, in our case).

The other aspect of effectiveness is end-to-end effectiveness. Given the emphasis on early precision in the context of web search, we measured NDCG at cutoff n [10], a well established metric for these types of search tasks. For the second stage reranker we used the linear model described above, whose quality on our limited feature set is on par with state-of-the-art tree-based methods.

5. RESULTS

5.1 Effectiveness

The relative recall of our approximate postings list intersection algorithm with various settings of r (bits per posting) and k (number of hash functions) is shown in Table 1 on the TREC 2009 web track queries (for a candidate list containing the top 10000 hits sorted by spam score).

These results are exactly what we would expect: Increasing the number of hash functions k reduces the false positive rate and hence improves recall. Increasing the number of bits per element r reduces hash collisions, thereby reducing the false positive rate and increasing recall. There is, of course, no free lunch: larger values of r increase memory requirements and larger values of k reduce speed.

For the end-to-end evaluation, output of the candidate generation phase (by our approximate postings list intersection algorithm and the exact baseline) were reranked by the linear machine-learned model described in Section 3. For each value of r and k in Table 1, we compared the two outputs in terms of NDCG@$\{1, 3, 5, 10, 20\}$. We omit the results here in the interest of space, but Wilcoxon tests (with $p = 0.05$) showed no significant difference in NDCG values.

These results empirically validate the assumption that underlies our work: that modern machine-learned ranking functions are sufficiently powerful to deliver high quality results as long as they are presented with a "reasonable" set of candidate documents. This is particularly true with the emphasis on early precision in web search—for example, to obtain a perfect NDCG@1 score, the ranker simply needs to identify *one* relevant document (of the highest relevance grade). Therefore, the fact that the relative recall of our approximate postings list intersection algorithm isn't perfect has no statistically significant impact on end-to-end NDCG.

$r \backslash k$	1	2	3
8	5.35 ($\pm$0.26)	7.73 ($\pm$0.37)	8.53 ($\pm$0.24)
16	4.94 ($\pm$0.21)	6.08 ($\pm$0.38)	6.41 ($\pm$0.39)
24	4.40 ($\pm$0.11)	5.09 ($\pm$0.13)	5.46 ($\pm$0.25)

Table 2: Average query latency (ms) for the AOL queries across 10 trials, with 95% confidence intervals.

5.2 Efficiency

Average query latency of our approximate postings list intersection algorithm on the AOL queries is shown in Table 2, for different values of r (bits per posting) and k (number of hash functions). Reported values represent the average across 10 trials for each parameter setting, along with the 95% confidence interval. Our best configuration is $r = 24, k = 1$, with an average query latency of 4.4ms; this achieves 93.96% relative recall compared to the baseline (see Table 1). If a higher relative recall is desired, $r = 24, k = 2$ is a good option, at 5.09ms per query and a relative recall of 98.04%. However, since in all our effectiveness experiments, end-to-end NDCG was statistically indistinguishable from the exact baseline, there does not appear to be a downside to simply selecting the fastest configuration.

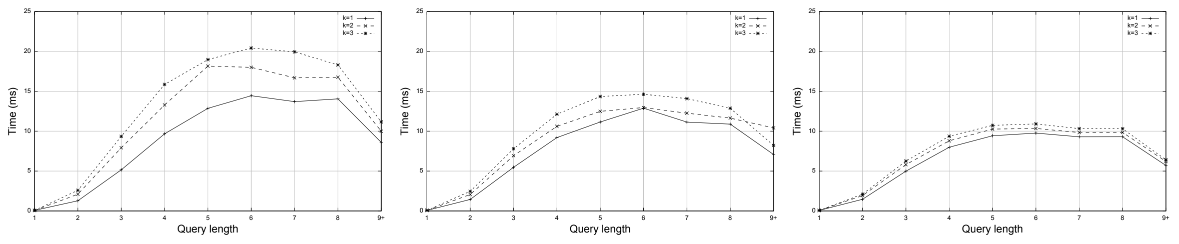
As we would expect, increasing k increases average query latency, since it requires computing more hash values and probing additional bit positions. Increasing r, however, calls into play two counteracting factors. On the one hand, increasing r leads to larger Bloom filters and hence less locality, which translates into more cache misses and longer memory latencies. However, on the other hand, as the size of the Bloom filter increases the false positive rate drops, and therefore fewer hash computations are needed to reject a non-existent document id. Empirically, we see that the second effect is stronger for the range of r values we explored: $r = 24$ yields the fastest speed for all values of k.

Figure 1 show average query latency for the AOL queries, broken down by query length (i.e., number of query terms after tokenization, stopword removal, etc.). As expected, query latency increases for longer queries, due to the need for more Bloom filter membership tests. However, beyond a certain point, latency levels off and even drops due to the appearance of rare terms in longer queries. With conjunctive query processing, in the worst case we need to only traverse the postings list of the least frequent term and can remove a document from consideration as soon as the membership test for a query term fails.

A final interesting observation: as r becomes larger, running time becomes less sensitive to k. In other words, increasing k increases query time less with larger values of r than with smaller values of r. This is because greater r reduces the probability of hash collisions, and our Bloom filter probing method early exits as soon as it finds an unset bit.

As a reference, the average query latency for exact postings list intersection using the small adaptive algorithm was 87.3ms on the AOL queries (for 10000 hits). Not only is small adaptive much slower overall, but query latency increases with longer queries. As discussed earlier, we put in a best faith effort to optimize our implementation, so we are confident that this represents a fair comparison.

We believe that one contributing reason for the slow speed of small adaptive is the amount of PForDelta decoding that



(a) $r = 8$ (b) $r = 16$ (c) $r = 24$

Figure 1: Effect of query length on latency using the AOL queries.

it must perform. In our Bloom filter algorithm, we only need to decode postings corresponding to the least frequent query term; and we only need to decode as many blocks as is necessary to accumulate 10000 candidate documents. The small adaptive algorithm, on the other hand, needs to decode postings for all query terms—and because PForDelta is blocked-based, probing an element (i.e., in binary search) requires reconstructing document ids for the entire block. As a result, small adaptive is highly dependent on the raw decoding speed of the PForDelta implementation, and this may be where language choice makes a significant difference. In all our algorithms we use the Kamikaze package for PForDelta compression, which was released by LinkedIn and represents an "industrial-strength" implementation. While it is perhaps true that a C/C++ reimplementation of our techniques may be faster, we are confident that the relative performance differences will hold, since Bloom filters will also benefit from a more efficient implementation.

Furthermore, we note that the small adaptive algorithm has aspects that are not cache friendly. Although there is definitely a dominant memory access pattern as postings are consumed, the binary search for locating the eliminator suffers from poor locality: subsequent probes move in unpredictable directions (thus, difficult to pre-fetch), not to mention further slowdowns by branch mispredicts.

Our third dimension of evaluation is index size, which is the amount of space required by the Bloom filters. As a reference, for the first segment of ClueWeb09 (around 50 million pages), the base postings lists (document ids only) total 13.88 GB in size. For our postings list intersection algorithm to work, we need additional space for the Bloom filters: with $r = 8$, an additional 11.67 GB; $r = 16$, 20.24 GB; $r = 24$, 27.40 GB. In terms of modern server configurations, these requirements are quite reasonable—in a mid-range commodity server today, one might expect 64GB RAM, which is more then sufficient to serve document partitions of the size we consider here.

6. CONCLUSION

This work tackles the problem of candidate generation in a two-phase retrieval architecture. Since powerful machine-learned rerankers are typically applied in the second phase, end-to-end retrieval effectiveness can be maintained as long as the candidate generation phase returns "reasonable" results. Thus, we can "cut corners" with approximate algorithms to speed up candidate generation—we propose a novel postings list intersection algorithm based on Bloom filters to accomplish this. Experiments showed that the algorithm is much faster than the small adaptive baseline, and yields end-to-end NDCG measures that are indistinguishable from those generated with the exact algorithm.

7. ACKNOWLEDGMENTS

This work has been supported in part by NSF under awards IIS-0916043 and IIS-1144034. Any opinions, findings, conclusions, or recommendations expressed are the authors' and do not necessarily reflect those of the sponsors. The first author's deepest gratitude goes to Katherine, for her invaluable encouragement and wholehearted support. The second author is grateful to Esther and Kiri for their loving support and dedicates this work to Joshua and Jacob.

8. REFERENCES

[1] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. *Workshop on Experimental Algorithms*, 2006.

[2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[3] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang. On the false-positive rate of Bloom filters. *Information Processing Letters*, 108:210–213, October 2008.

[4] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. *CIKM*, 2003.

[5] C. Burges. From RankNet to LambdaRank to LambdaMART: An overview. Technical Report MSR-TR-2010-82, Microsoft Research, 2010.

[6] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at Twitter. *ICDE*, 2012.

[7] G. Cormack, M. Smucker, and C. Clarke. Efficient and effective spam filtering and re-ranking for large web datasets. *CoRR*, abs/1004.5168, 2010.

[8] E. Demaine, A. López-Ortiz, and J. Munro. Experiments on adaptive set intersections for text retrieval systems. *Workshop on Algorithm Engineering and Experiments*, 2001.

[9] Y. Ganjisaffar, R. Caruana, and C. Lopes. Bagging gradient-boosted trees for high precision, low variance ranking models. *SIGIR*, 2011.

[10] K. Järvelin and J. Kekäläinen. Cumulative gain-based evaluation of IR techniques. *ACM TOIS*, 20(4):422–446, 2002.

[11] H. Li. *Learning to Rank for Information Retrieval and Natural Language Processing*. Morgan & Claypool, 2011.

[12] J. Li, B. T. Loo, J. Hellerstein, M. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. *Workshop on Peer-to-Peer Systems*, 2003.

[13] J. Lin, D. Metzler, T. Elsayed, and L. Wang. Of Ivory and Smurfs: Loxodontan MapReduce experiments for web search. *TREC*, 2009.

[14] D. Metzler. Automatic feature selection in the Markov random field model for information retrieval. *CIKM*, 2007.

[15] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. *Conference on Scalable Information Systems*, 2006.

[16] M. Shepherd, W. Phillips, and C.-K. Chu. A fixed-size Bloom filter for searching textual documents. *The Computer Journal*, 32(3):212–219, 1989.

[17] S. Tatikonda, B. Cambazoglu, and F. Junqueira. Posting list intersection on multicore architectures. *SIGIR*, 2011.

[18] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *VLDB*, 2009.

[19] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. *ICDE 2006*, April 2006.