Fast First-Phase Candidate Generation for Cascading Rankers

Qi Wang
Computer Science & Eng.
New York University
qiwang@nyu.edu

Constantinos Dimopoulos
Computer Science & Eng.
New York University
constantinos@nyu.edu

Torsten Suel
Computer Science & Eng.
New York University
torsten.suel@nyu.edu

ABSTRACT

Current search engines use very complex ranking functions based on hundreds of features. While such functions return high-quality results, they create efficiency challenges as it is too costly to fully evaluate them on all documents in the union, or even intersection, of the query terms. To address this issue, search engines use a series of cascading rankers, starting with a very simple ranking function and then applying increasingly complex and expensive ranking functions on smaller and smaller sets of candidate results. Researchers have recently started studying several problems within this framework of query processing by cascading rankers; see, e.g., [5, 13, 17, 51].

We focus on one such problem, the design of the initial cascade. Thus, the goal is to very quickly identify a set of good candidate documents that should be passed to the second and further cascades. Previous work by Asadi and Lin [3, 5] showed that while a top-k computation on either the union or intersection gives good results, a further optimization using a global document ordering based on spam scores leads to a significant reduction in quality. Our contribution is to propose an alternative framework that builds specialized single-term and pairwise index structures, and then during query time selectively accesses these structures based on a cost budget and a set of early termination techniques. Using an end-to-end evaluation with a complex machine-learned ranker, we show that our approach finds candidates about an order of magnitude faster than a conjunctive top-k computation, while essentially matching the quality.

1. INTRODUCTION

Search engines are continuously optimizing their ranking functions in order to improve result quality. This is usually achieved through more and more complex ranking functions based on large sets of features, including features derived from text, link structure, past queries, and online or proprietary data sets and knowledge bases through various data extraction and mining techniques. However, these complex ranking functions create significant performance challenges for the engines, as evaluating them on large numbers of documents is very expensive. Given the billions of queries that have to be processed each day, it would not be feasible to apply such functions directly on all documents passing the initial Boolean filter, even in the conjunctive case.

To address this challenge, all current major engines appear to be using a cascading approach to query processing that approximates the results of such ranking functions through a series of increasingly more complex intermediate functions. Thus, after query analysis and rewriting, the search engine first applies a very simple ranking function, similar to BM25 or a Cosine measure, that only uses a few features and can be very efficiently applied on large numbers of candidates. We refer to this process as the first cascade. Next, in the second cascade, a somewhat more complex function based on a larger set of features is applied to, say, the top few thousand results from the first cascade. In the third and further cascades, even more complex functions are applied to smaller and smaller sets of surviving candidates of the previous cascades. This approach was described and formalized in [51].

Thus, the goal of the cascading approach is to return (almost) the same results as we would get from applying the complex function on all candidates, at a fraction of the computational cost. Its proper implementation, however, poses several challenges that have recently received some attention in the literature [5, 13, 17, 32, 40, 51]. In particular, work has focused on four distinct challenges: (1) How to design good sequences of increasingly complex ranking functions and associated cutoffs (number of results kept for the next cascade) [51]; (2) how to efficiently apply a complex ranking function to candidates by using early-exit strategies [13]; (3) how to design ranking functions for the first cascades that preserve many good candidates for subsequent cascades, as opposed to focusing on how to order a few top results [17]; (4) how to implement the first cascade efficiently through an optimal choice of Boolean filters and various early-termination techniques [3, 5].

We focus on the last challenge, which is important as the first cascade is executed on large numbers of candidates. The results in [3, 5] indicated that for the first cascade, a conjunctive filter does essentially as well as a disjunctive one, while saving a lot of time. However, a naive attempt to use a global document ordering to avoid a complete conjunctive traversal of the index structures resulted in significant losses in end-to-end result quality.

In this paper, we propose an approach that runs about an order of magnitude faster than even highly optimized con-

SIGIR '16, July 17-21, 2016, Pisa, Italy
@ 2016 ACM. ISBN 978-1-4503-4069-4/16/07 $15.00
DOI: http://dx.doi.org/10.1145/2911451.2911515.

junctive and disjunctive top-k computations, while achieving essentially the same end-to-end result quality. More precisely, given a complex ranking function that needs to be approximated by a cascading approach, an inverted index structure for a document collection, and a training query trace, we show how to build an auxiliary layer of index structures, and how to select which parts of this layer to consult on a submitted query, in order to obtain high-quality candidates with a limited computational and space budget.

In particular, our contributions are as follows:

1. We design a framework that builds specialized single-term and pairwise index structures subject to a given space constraint. It uses limited size query traces to train query term language models and posting quality models.
2. We propose an online selection algorithm based on a cost budget that, for a given query, decides the access depth for each available structure according to the posting quality model, and present query processing and lookup strategies that further improve performance.
3. We provide an end-to-end evaluation of our proposed architecture on ClueWeb09B, and show that our approach can identify candidates about an order of magnitude faster than previous published results, with negligible quality loss.

The remainder of the paper is organized as follows: Section 2 presents some background and discusses related work. In Section 3 we give a high-level overview of our approach, and present our solutions for several components. Next, Section 4 outlines the experimental setup, and Section 5 presents the experimental evaluation of the proposed framework. Finally, Section 6 provides some concluding remarks.

2. BACKGROUND AND RELATED WORK

In this section, we provide some background on inverted indexes, query processing and early termination techniques, and cascade ranking architectures, and discuss related work.

Inverted Indexes: Commercial search engines perform query processing based on the widely used inverted index [58]. Given a collection of N documents, each document is assigned a unique identifier (docID) from 0 to N-1. The inverted index consists of a set of inverted lists and a lexicon. In particular, for each distinct term t in the collection, there is an inverted list L_t. Each L_t is a list of postings specifying the documents that t appears in. Typically, each posting contains the docID of a document containing t and the frequency of t in the document; however, there may also be other information, such as the positions of the term occurrences in the document, or a precomputed impact score. The lexicon contains for each unique term in the collection a pointer to the corresponding inverted list. Inverted index compression is crucial for search engine performance and many techniques have been proposed [6, 45].

Index Layout: There are many ways to organize the inverted lists. In document-sorted indexes, each list is sorted by docID, resulting in small delta gaps (d-gaps) between consecutive docIDs that lead to a smaller compressed size [45]. Impact-sorted indexes sort the postings in each list in decreasing order of impact scores. Thus, high-scoring query results tend to be located near the front of the lists, potentially enabling a smart query processing algorithm to skip most of the rest of the list. However, this approach leads to poor compressibility compared to document-sorted indexes, and may require random lookups into lists for docIDs that score high on one query term but low on others. Finally, impact-layered indexes split each inverted list into a small number of layers based on impact scores. Our approach uses two layers, where the first layer of high-impact postings is sorted by impact, and the second layer by docID to allow for efficient random lookups.

Index Traversal: During query processing, index structures can be traversed in different ways [50]. In Document-at-a-time (DAAT), each list has a pointer to a current posting and one document is processed at a time; then pointers are moved forward in docID space. The top results are usually maintained in a min-heap structure. In Term-at-a-time (TAAT) traversal, a list is fully traversed before accessing the next one. Partially scored documents are kept as accumulators in a hash table or other structure; this structure can be a bottleneck if it grows beyond the CPU caches. Lastly, there are hybrids between DAAT and TAAT. Note that DAAT is mainly suitable for document-sorted indexes, while TAAT works well with impact-sorted ones.

2.1 Query Processing and Early Termination

The simplest form of query processing applies a Boolean filter (AND/OR) on the inverted lists of the query terms, and then ranks all documents passing the filter. A good ranking function should (a) provide a good approximation of the relevance of a document with respect to a query and (b) be efficiently computable using the information stored in the index. Well-known examples include Cosine measures and BM25 [6]. Most of these simple ranking schemes have the property that the score of a document d for the full query q is the sum (or other simple combination) of per-term scores; i.e., $score(q,d) = \sum_{t \in q} s(t,d)$, where $s(t,d)$ is the impact score of term t in d.

Early Termination: A naïve query processing approach is inefficient, and ends up decompressing and accessing large parts of the inverted lists. To improve on this, researchers have proposed many *early termination* (ET) algorithms that try to find good results while accessing and scoring only a small part of the relevant inverted lists. ET algorithms are called *safe* if they always return the same results as an exhaustive algorithm, and *unsafe* otherwise [48]. ET techniques are widely used in commercial engines and academic systems, and include the following approaches:

- **Index Tiering:** A collection is partitioned into, say, 2 or 3, disjoint subsets of documents called *tiers*, where the first tier contains the highest-quality documents; queries are executed on the first tier and only selectively routed to other tiers [41, 42].
- **Pruning:** In static pruning, postings considered unlikely to ever be useful are deleted from the index [8, 12, 24, 37]. In dynamic pruning, inverted lists are typically organized in impact-sorted or impact-layered form, and algorithms focus on high-impact postings and only selectively access lower-impact postings for promising documents. One very widely studied example are the FA, TA, and NRA algorithms in [23].
- **Skipping:** For document-sorted indexes, there are various techniques for skipping unimportant parts of the inverted lists [10, 14, 19, 20, 21].

Our approach here is unsafe, and based on dynamic pruning with impact-layered structures for both terms and term pairs (intersections of two terms), as described later.

2.2 Cascading Ranking Architectures

In modern commercial engines, query processing is based on cascading ranking schemes [10, 13, 51], where each cascade includes a ranker that provides candidate documents to subsequent cascades. The first cascade is usually based on a very simple ranking function that is evaluated on large parts of the index structure to get an initial set of candidates. Thus, this function must be very fast, while providing a reasonably high-quality set of candidates. Subsequent cascades are executed on fewer candidates using more complex and expensive ranking functions. The challenge in designing such cascading architectures is to select a set of cascades and associated ranking functions that achieves high end-to-end quality at low cost.

Cascading setups are crucial for the performance and quality of modern commercial engines, and a number of papers have recently focused on this setup [4, 5, 13, 17, 32, 40, 51]. We focus on optimizing the efficiency of the first cascade in such architectures, a problem recently studied in [4, 5].

2.3 Comparison to Previous Work

We now discuss the relationship of our approach to previous work. The particular problem we consider is based on the setup in [5, 4, 17]. Thus, we have a two-phase cascading architecture, where the first phase obtains an initial candidate set of, say, several hundred or thousand documents, while the second phase reranks these candidates using a more sophisticated ranker based on dozens or hundreds of features. Our goal is to design very fast ET algorithms for the first phase that achieve end-to-end quality comparable to more exhaustive approaches. This is essentially the problem addressed by Asadi and Lin in [4, 5].

In particular, [5] investigates the efficiency/effectiveness trade-off for various first-phase candidates generation approaches. They experiment with conjunctive WAND [10], disjunctive WAND, and two conjunctive algorithms that first obtain the intersection and then rerank results based on BM25 or spam score, and conclude that conjunctive WAND provides the best trade-off. Work in [4] shows how to accelerate the intersection-based approaches using Bloom Filters. Our contribution here is to provide a method that achieves quality comparable to their best methods at lower cost.

Another relevant recent work [52] proposes a *document prioritization method* for selective evaluation of documents that achieves a better efficiency/effectiveness balance in the first cascade. The running times reported in [52] are significantly slower than ours, though some of the ideas could potentially be used to further optimize the lookup phase of our approach.

Our algorithm is based on a layered index organization and performs a limited-depth access to impact-sorted single-term and term-pair structures, followed by random lookups. As such, it is closely related to the well-know FA algorithm proposed by Fagin [23], and also to ET algorithms for impact-layered indexes introduced in [39]. There are many subsequent papers that further developed and often combined these approaches to solve various IR ranking problems, including, e.g., [2, 7, 9, 33, 47, 48].

We note that [28, 29] describe a number of access and lookup strategies for top-k query processing in database applications. One algorithm that is somewhat similar to our approach is the *MPro* algorithm in [28], which seeks to minimize lookup costs through sorted access. Work in [48] suggested methods for selecting the access depth into the available single-term impact-sorted lists. However, [48] focused only on single-term impact-sorted lists and did not provide cost-based query processing algorithms in the context of cascading rankers. Our approach is different from both of the above as we are proposing a framework for constructing additional impact-sorted index structures, including pairwise structures, based on query term distributions and posting quality models and subject to space constraints, plus an online depth selection algorithm and lookup strategies.

Our approach relies heavily on term-pair index structures, introduced in [34] and subsequently studied in a number of papers such as [11, 15, 26, 30, 44, 54, 56, 57]. Our work is most closely related to [30], which also applies the approach in [23] to pairwise structures. The main difference is our focus on cascading ranking schemes, and our framework for optimizing index structures and index traversals based on a limited space and access cost budget.

Also related is the work in [11], which proposes building single and pairwise impact-sorted lists that are then completely accessed during query processing. Though related, our work is different in two ways: While [11] and earlier work in [44, 54, 56, 57] assume a ranking function that directly takes proximity into account, we assume a more complex function in a cascading setup. Also, the resulting auxiliary structures in [11] are much larger than the basic index size; in contrast, our query language and posting quality models allow us to achieve high speed with only a limited increase in size. We note that our results could potentially be improved by adding special pairwise structures for terms occurring close to each other in a document, as done in [11, 54].

Finally, [1, 27] study techniques for learning better index structures given a set of documents and queries. In particular, [27] can be seen as essentially learning an ordering of index postings that is better than the "natural" impact-based ordering used, e.g., in [23]. We note that this issue is orthogonal to our approach, and could be combined to possibly yield additional benefits. Lastly, while we construct our first-layer structures using off-line preprocessing, one could also approach this via a suitable caching mechanism, such as those in [25, 38, 46] for other types of structures. Finally, our improvements are in addition to any speedups achieved through result caching, since our query traces do not have significant numbers of repeated queries.

3. PROBLEM SETUP AND APPROACH

We now define and discuss our problem setup, give a high-level description of our approach, and then provide more details about the various steps that are involved.

3.1 Problem Setup

We are given a complex ranking function CF, a simple ranking function SF used as the first cascade, and a rank cutoff c for the first cascade, meaning that only c results from the first cascade will be evaluated by the complex ranking function, which will then return the top k, $k \leq c$, results to the user. Our goal is to implement the first cascade to run as fast as possible without significantly decreasing end-to-end result quality. In our implementation, we allow unsafe early

termination techniques, that is, the c results we give to CF may be different than those obtained from an exhaustive top-c computation using SF.

We measure quality in two ways: (i) *Overlap@(k,c)*, meaning how many of the top-k results that would be returned by an exhaustive application of CF (i.e., $c_{cf} = \infty$ or at least fairly large) are returned with our first cascade implementation that evaluates c candidates and (ii) *NDCG@k*, which is the normalized discounted cumulative gain that considers the order of the results.

Problem Discussion: Note that while the above definition assumes only two cascades, SF and CF, this does not really limit our approach as long as any additional intermediate cascades do a good job at approximating CF, i.e., do not lose too many good results among those nominated by SF for further processing. We believe this is a reasonable assumption in practice, and assume that subdividing the second cascade into further cascades is a separate problem. For the same reason, the reported running times are only for the first cascade, as the cost of the second cascade should only depend on c.

While an unsafe implementation of the first cascade could in principle achieve better quality than a safe disjunctive or conjunctive top-c computation using SF, this is not really expected. Thus, our goal is to do (almost) as well as these two choices, shown to be good in [3, 5], while being much faster than these and other non-safe competitors.

3.2 Our Overall Approach

We now describe our approach, which starts with an existing inverted index for the collection that could be used to run queries using SF. We then create additional auxiliary index structures to quickly identify promising candidates. We create two kinds of structures, single-term structures, and term-pair, or pairwise, structures, which together make up the *first layer* of the index, while the complete inverted index[1] makes up the second layer. For the single-term structures in the first layer, we choose, for each inverted list, some number of high-scoring postings, and arrange them by impact score in decreasing order. The pairwise structures are obtained by intersecting two inverted lists, and keeping a certain number of high-scoring two-term postings, where the score of such a posting is the sum of the impact scores of the two constituent postings. These structures are also sorted in decreasing order of impact score. We later discuss how to select which postings to put into the first layer, based on query traces and impact scores.

When a query enters, the first cascade is now executed by first selecting and accessing some prefix of the much smaller relevant structures in the first layer of the index. Afterwards, we perform a limited number of lookups into the second layer, to obtain additional scores for some promising documents for which we have found partial scores in the first layer. Finally, we identify c documents for further evaluation by CF. We show the overall index structure in Figure 1, where a query "dog cat mouse" is processed.

For our problem setup, there are various technical problems to address. In particular, when building the first layer, we need to decide how deep we should build the single-term structures, and for which pairs of terms we should build a pairwise structure and up to what depth. The goal is for the structures to be deep enough to find most good results,

[1]Except for short lists that are completely in the first layer.

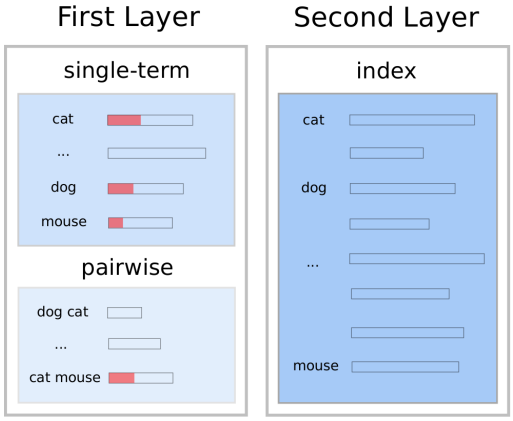

Figure 1: Our index structure, with first layer on the left and second layer on the right. In the top left are single-term structures, and in the bottom left are pairwise structures, each sorted by decreasing impact score. For a query "dog cat mouse" our method might decide to access a certain prefix (shown in red) of each relevant single-term list, and of one of the available pairwise structures (in this case, for "cat" and "mouse"), based on some access budget.

but not so deep that there is a large increase in overall index size. When a query enters, we need to decide which of the applicable index structures to consult and up to what depth – always using all potentially relevant structures in the first layer up to their full depth would not be efficient. We also need to decide what lookups into the second layer should be performed to identify the c results to be forwarded to the complex ranker for full evaluation. We later propose and evaluate solutions for all these problems.

Overall, our approach has the following steps that need to be implemented. During indexing, we have two steps:

- **Modeling:** We build two types of models by performing training on a query trace of limited size, (a) a language model for the queries that allows us to predict how frequent certain terms and combinations of terms are in the query trace, and (b) a quality model relating the rank of a posting in a first-layer structure to its likelihood of being a top result under CF.
- **Index Building:** We build the first layer based on the constructed models, by carefully choosing which structures to build and up to what depth, subject to a maximum space budget.

Later, when a query enters the system, we execute the following sequence of steps:

- **Online Greedy Depth Selection:** We consider the list of relevant structures, and decide which of them should be accessed and to what depth, based on the quality model and based on a simple model for query processing costs. In fact, costs will be modeled based on the aggregate access depth into the first-layer structures and the number of lookups into the second layer, with a certain budget available for a query.
- **Query Processing:** We throw the accessed structures and their corresponding access depths, as selected in the previous step, into a simple but fast in-memory query processor.

- **Second-Layer Lookups:** We decide for which candidate documents we should perform lookups into the second layer to get more precise scores.
- **Final Selection:** We choose the c results that should be evaluated by the complex ranker.

3.3 Index Construction

We now describe the two steps in the index construction in more detail. First, we build two models, one for query and query term distribution, and one to model the quality contributions of different parts of the index structures. These models are then stored for later use during index building and query processing.

Modeling: For the first model, we use standard Language Modeling tools, in particular the MIT Language Modeling (MITLM) toolkit[2], based on Kneser-Ney smoothing. We train these models on part of our query trace (distinct from any queries used in the evaluation), to obtain estimates of two probabilities, $p(t)$, the probability that term t occurs in a random incoming query, and $p(t_1, t_2)$, the probability that both t_1 and t_2 occur in such a query. We refer to these as our *query language models*.

For the second model, given a posting p for a term t that has rank r in its list (i.e., has the r-th highest impact score in its list), we want a rough estimate of the likelihood that the posting belongs to a top-k result under the complex ranker CF, given a random query containing t. This is done by issuing training queries and, for each posting in one of the query term lists, storing its rank, the length of its inverted list, and whether it is part of a top-k result for the query. We bucketize the list lengths and relative ranks within lists into dozens of ranges (classes) each. Then we aggregate our data into a two-dimensional table A where $A[i, j]$ estimates the probability that a posting belonging to list length class i and relative rank class j (which might, say, correspond to a list length between 1000 and 1500 and rank between 120 and 160) leads to a top-k result. This approach gives sufficiently accurate predictions, while allowing extremely fast lookups during index building and query processing to get an estimate of $p(\text{top-k}|t)$, the likelihood that a posting is part of a top-k result given that its term t is part of a query.

We then repeat the process for term intersections, where we create a table where for each posting in the intersection of two terms t_1 and t_2, we use the length of the intersection, and the rank of the posting in the intersection, to estimate $p(\text{top-k}|t_1, t_2)$, the likelihood that a pairwise posting is part of a top-k result given that both t_1 and t_2 occur in a query. We refer to these models for single lists and term intersections as *posting quality models*.

Index Building: Given a space budget, our next goal is to build a first layer containing term and term-pair postings that are likely to lead to top-k results under random queries. To do this, we allocate separate space budgets to the single-term and pairwise structures in the first layer. For single-term lists, we greedily pick postings from the highest ranks of the inverted lists to add to the first layer. That is, we try to pick postings with the highest value of $p(t) \cdot p(\text{top-k}|t)$. Since our estimate for this value based on the models is expected to be a monotonically decreasing step function in each list, we can sort each list by increasing rank, and select chunks of postings with equal value from the beginning of the lists until the budget is exhausted using a heap to decide from which list to pick.

We repeat this greedy selection process for pairwise postings. Since there is a huge number of term pairs, we first restrict the space by only considering intersections for terms t_1 and t_2 with $p(t_1, t_2) \geq \theta$ for some small θ. Then these intersections are created and sorted by impact, and we again select chunks of postings from the beginning of the intersections based on our estimates of $p(t_1, t_2) \cdot p(\text{top-k}|t_1, t_2)$, until the budget is exhausted.

All structures in the first layer are kept sorted from highest to lowest impact score. The single-term postings are of the form (docID, impact), while the term-pair posting layout is (docID, impact_{t_1}, impact_{t_2}). Note that for single-term structures, we do not remove the postings in the first layer from the second layer, but create a copy of the postings, so this uses extra space. The reason is that we only access a limited amount of the first-layer structure, and thus we need to make sure that a lookup into the second layer can retrieve all postings. An exception are very short lists, of length less than 100, where we always move the entire list into the first layer and access it fully on any query containing the term; thus, these lists do not increase space usage (though their overall size is small). We also added an additional rule that limited the depth of any selected single-term and pairwise structure to the maximum access depth for queries, typically several thousand postings, as any posting deeper than the access depth would never be used anyway.

3.4 Query Processing

We now describe the steps involved in query processing: the online greedy depth selection, the query processor, the second-layer lookups, and the final selection of candidates.

Online Greedy Depth Selection: Given a query, we first identify all relevant structures available in the first layer. This usually includes all single-term structures for the query, since our language model assigns non-zero probabilities to all terms and the first postings tend to have very high values of $p(\text{top-k}|t)$, especially for short lists. Only some of the pairwise structures will typically be available for a query.

For each query, we have a cost budget that determines how much of the relevant structures we can access. For example, we might have a budget $b = 1000$, meaning that we can only access a total of b postings from the structures. Then for each structure we select a (possibly empty) prefix of postings. This is done greedily using a heap, as during index building, except that we select chunks of postings based on $p(\text{top-k}|t)$ and $p(\text{top-k}|t_1, t_2)$, respectively, without multiplying by $p(t)$ and $p(t_1, t_2)$ (since at this point the query already contains the terms). We note that we could perform various refinements to this approach, by charging different costs for pairwise versus single-term postings, or assigning different budgets to queries based on their difficulty.

Query Processor: We now run a fast and simple query processor on the selected structures and their corresponding depths. This processor copies the selected prefixes of the first-layer structures into an array and then runs a fast Radix Sort to sort postings by docID. A subsequent scan then aggregates the impact scores for each docID, and creates a bit filter for each docID stating which terms may require lookups into the second layer. During the scan we also filter redundant lookups as follows: Suppose a posting with docID 7 and impact 2.9 was found in the prefix of the list

[2]Available at `https://code.google.com/p/mitlm/`

for "cat", and that we have also accessed all pairwise postings for the pair "cat dog" with score 2.1 or higher. Then we do not have to perform a lookup into the "dog" list in the second layer for docID 7 – if such a posting existed we would have seen it as part of a pairwise posting.

We initially implemented a TAAT query processor using a hash table for the accumulators. However, we found that the sorting-based approach was much faster, by a factor of 2 to 3. Such a sorting-based approach is possible because we fix the access depth for each structure at the start of the query.

Second Layer Lookups and Final Candidate Selection: Next, we check the candidates and their accumulated scores, where most of these scores are partial, and many may have been only seen for one of the query terms. We now decide for which of these candidates we should perform lookups into the second layer to complete their scores, subject to a budget on the number of lookups (say, a few thousand per query). This is done using the accumulated partial score, as this provides a strong signal for relevance. We select the candidates with the highest partial scores, by running a randomized approximate selection algorithm where we first draw a sample of the impact scores, sort this sample, pick a suitable threshold from the sample, and then keep all candidates with impact above this score. Finally, we perform all necessary lookups for these candidates into the second layer, and keep the c candidates with highest completed BM25 scores, to be submitted to CF.

4. EXPERIMENTAL SETUP

In this section, we describe the data sets, ranking models, evaluation metrics, and setup of our experiments.

Datasets: All our experiments were conducted on the ClueWeb09B collection, which consists of $50,220,423$ documents, $86,532,822$ distinct terms, and 17 billion postings. For evaluation, we used the TREC 2009 Million Query Track (40k), which we refer to as Million09. Our training set for the modeling step includes 30k queries selected at random from Million09, while the testing set for performance evaluation consists of 3k from the remaining 10k queries. Table 1 shows the query lengths for the testing query set.

Query length	2	3	4	$\geq$5
# queries	1408	954	517	121

Table 1: Query length distribution for the 3k testing queries.

The TREC 2010 to 2012 Web Track topics (150) were used for training the machine-learned complex ranker CF. For the language models, we used linear interpolation of a model for the training queries of the query set with a model for a randomly selected sample of 1.5 million documents, using the MIT Language Modeling (MITLM) toolkit.

Ranking models: We selected the BM25 ranking scheme as our first ranker, as it is widely used as a simple ranker in the literature and satisfies the desired properties of being both computationally fast and providing a reasonable set of initial candidates.

Recent studies [31, 51] show that ranking schemes obtained using learning-to-rank methods with dozens or hundreds of features outperform traditional bag-of-words models in terms of quality. There are a number of learning-to-rank tools that are available. We decided to use LambdaMart [53] to learn our complex ranker CF, as it is considered one of the most effective learning-to-rank models [22, 35]. We trained on the 150 queries from TREC 2010 to 2012 based on standard features from the literature [5, 36, 49]. Table 2 lists a subset of these features. The anchor text features were extracted using the data from [18], while the spam and pagerank values are from [16]. The distribution feature refers to the dispersion of term occurrences across a specific document, when the document is split into pieces of fixed size, say 100 terms.

Evaluation metrics: The main aspects in the design of a scalable web search architecture include quality, time, and space. Thus, we evaluated the proposed framework on these three aspects. We measure the end-to-end quality of the proposed methods with Overlap@(k,c) and NDCG@k. In particular, the end-to-end effectiveness evaluation within the cascading ranker setup is performed as follows. In the first cascade, the top-c documents are obtained based on our method and then, in the second cascade, the CF is applied to these documents in order to return the final top-k.

For the effectiveness of the first cascade, we measure Overlap@(k,c) as the fraction of top-k documents obtained when applying CF to all top 2000 results of a safe disjunctive BM25, that are also found among the c candidates computed by our algorithm. While it was not feasible to apply the complex ranker to all documents in the union of query terms, we found in preliminary experiments that beyond the top 2000 there was little change in the final top-k. This choice is also directly supported by the recommendations in [35]. Thus, an overlap of 1.0 for a query means that all top-k results were found among the c candidates of our method. We use $k = 10$ unless stated otherwise.

We evaluated the speed of our methods using average query latency (in milliseconds) for generating the candidates. That is, we measure the time elapsed from when a query arrives until the time when the top-c candidates are ready to be evaluated by CF. (We do not count the time for applying CF as it is the same for all methods.) The space overhead was measured as the percentage of a baseline full index.

Index: The second layer was indexed and compressed using a version of PForDelta [59] proposed in [55]. The algorithms were implemented using C++ and compiled using gcc with -O3 optimization. The experiments were conducted on a single core of a 2.27Ghz Intel Xeon (E5520) CPU. All data structures and indexes are memory-resident.

Parameters: Overall, the proposed framework utilizes the following parameters: (a) the access cost budget, i.e., the number of postings to access per query, (b) the number of documents to perform lookups on, (c) the number of candidates to forward to CF, c, and (d) the space budget, i.e., the amount of additional space beyond a standard index. During selection of the term-pair structures in the first layer, we considered only pairs with probability (according to the language model of MITLM) at least $1.99 * 10^{-16}$ for the Million09 query trace. Next, we present the experimental evaluation of our approach based on these parameters.

5. EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation of our methods in terms of effectiveness, efficiency, and space.

5.1 Without Space Constraints

In the first experiment, we evaluate the proposed candi-

Textual	BM25, language model, anchor text language model [36]
Positional	absolute first position, relative first position, distribution
Query-based	query size, fraction of numbers in query, list length
Dependency Models	language model of sequential and full dependent query terms [36]
Document-based	document length, document bytes, url length, url nesting, # outlinks, position in alexa, spam and pagerank [16]

Table 2: Categories of features utilized for CF training.

date generation algorithm under the assumption that there is no space budget; i.e., all possible first layer single-term and pairwise structures are available at query time. Although this scenario is unrealistic, as the space overhead of the pairwise structures can be very large, it shows the potential of the proposed method.

We compare our method against a naïve baseline which, given a cost budget b, accesses all relevant structures of the first layer at equal depth. Thus, if there are 5 available structures for a specific query, and the cost budget is $5k$ postings, the baseline would access the first $1k$ postings in each structure. Note that this approach is similar to the *Fixed* method proposed in [48]. We evaluate three versions of this naïve approach, for the cases where only single-term, only pairwise, and both types of structures are available, to show that both types of structures give benefit. Moreover, we implemented a *clairvoyant* selection algorithm that knows apriori which of the docIDs in the postings in the first layer result in top-k results, and then selects prefixes of the structures in an optimal way, thus giving an upper bound on the quality that can be achieved with any depth-limited access scheme on impact-sorted structures. This algorithm is included to observe how close to optimal our algorithm is.

Setup: We assume unlimited space budget and use the following parameter settings and selection strategies: c is 500, the algorithms exhaustively perform lookups on all docIDs seen in the accesses of the first-layer structures (thus the number of lookups is only bounded by the access depth), and the c candidates are selected based on highest BM25.

Effectiveness: Figure 2 shows Overlap@$(500, 10)$, i.e., the fraction of correct top-10 results preserved within the $c = 500$ candidates, as the access cost budget varies from 500 to 20000, and with the first layer consisting of different structures. Obviously, the clairvoyant algorithm achieves the best quality for all access budgets, as it is as an upper bound of our method. On the other hand, we observe that for the naïve method, having only pairwise structures consistently outperforms having only single-term ones, but having both achieves the best quality, which is 0.8864 when the cost budget is 2000. The Greedy selection algorithm outperforms all naïve methods and achieves quality close to the optimal clairvoyant one, even with moderate access budget. For instance, the quality is 0.946 for the 2000 access budget. Thus, more than 94% of the same top-10 results are returned. Finally, as the cost budget increases, quality increases for all algorithms, since we consider more postings as candidates. In particular, Greedy achieves quality really close to Clairvoyant for the 5000 access budget.

Efficiency: In Table 3, we report the average query processing time in milliseconds of the Greedy algorithm when the access budget varies from 500 to 20000. We observed that the access budget is a very good proxy for query processing time in the case of unlimited lookups, and thus we only report the time for Greedy; the numbers for the naïve and clairvoyant algorithms with the same access budget are very similar. According to Table 3, we see that it is possible

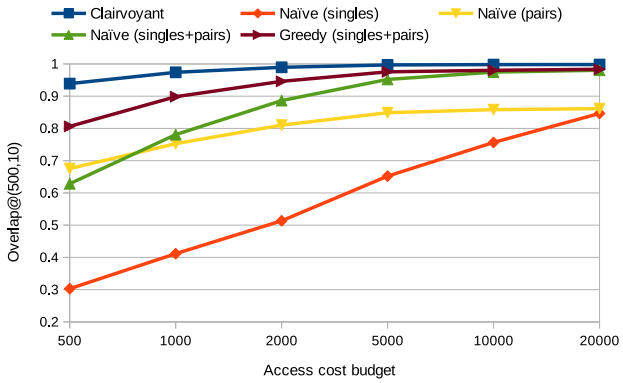

Figure 2: Effectiveness of the first-phase selection algorithms at various access cost budgets on Million09 queries, assuming no limit on space.

to achieve overlap close to 0.946 for access budget 2000, in 0.51ms. As the access budget increases, query processing becomes slower, as we process more postings. Thus, both quality and speed are very good when there is no space constraint. Next, we evaluate if comparable numbers can be obtained with limited space.

Access budget	500	1000	2000	5000	10000	20000
avg qp (ms)	0.22	0.33	0.51	0.95	1.6	2.79

Table 3: Efficiency of the Greedy algorithm with various access cost budgets on Million09 queries, assuming no limit on space.

5.2 With Space Constraints

In the next experiment, we drop the assumption of unlimited space budget and focus on more realistic scenarios. More specifically, we allow a specific percent of space overhead for the first-layer structures over the full second-layer index. Given various space budgets, we evaluate the Greedy algorithm in terms of quality and speed. In this experiment, we use the setup and parameters of the previous experiment, and assume that the first layer consists of both single-term and pairwise structures. We still do all lookups to complete the partial scores of all docIDs seen in the first layer.

Effectiveness: In Table 4, we present the effectiveness of the Greedy algorithm (measured by Overlap@$(500, 10)$) when a specific percentage of space overhead is allowed for the first layer structures, and the access cost budget is 2000 and 5000. For the single-term structures, we decided to keep up to 2000 and 5000 postings of every list, which correspond to 7.1% and 9.9% of the full index, respectively. On top of this small fixed space overhead, we have a limited space budget for pairwise structures that is allocated according to the Greedy allocation algorithm. The reported space in Table 4 includes only the pairwise structures in the first layer. First, we observe that our proposed method works quite well even with limited space budget, with moderate quality loss. (We look at NDCG numbers later.) Increasing

the space budget of pairwise structures to more than 50% does not seem to provide significant quality gains unless a lot of space is available. When the access cost is larger, again better quality is achieved as more candidates are considered.

Space	0.1	0.3	0.5	0.7	1	∞
2000	0.8093	0.8359	0.8412	0.8426	0.8428	0.946
5000	0.8717	0.895	0.9003	0.9028	0.9037	0.9755

Table 4: Effectiveness of the Greedy algorithm (Overlap@$(500, 10)$) at various space budget setups, for 2000 and 5000 access budgets on the Million09 query trace.

A space overhead of 57.1% (7.1% singles and 50% pairs) for 2000 access budget, and 59.9% for 5000 is acceptable given the significant performance speedup that we achieve. For example, in the Maguro system [42], a 20x index size increase is justified for a 3x performance improvement.

Efficiency: Table 5 shows the average query processing time of the Greedy algorithm for various space budget, when access budget is 2000 or 5000. As mentioned before, the access budget provides a reliable proxy for performance and this is evident in Table 5. The Greedy algorithm requires on average 0.551ms and 1.055ms, for 2000 and 5000 access budget, respectively, still assuming no limit on lookups. As the space budget increases, the performance for both access budgets becomes better. The reason is that more high quality postings appear in the pairwise structures, which results in fewer lookups for the missing terms.

Space budget	0.1	0.3	0.5	0.7	1	∞
2000	0.581	0.56	0.558	0.556	0.555	0.5
5000	1.125	1.08	1.065	1.055	1.049	0.957

Table 5: Efficiency of the Greedy algorithm with varying space budget, for 2000 and 5000 access budgets on the Million09 query trace, measured in ms.

5.3 Lookup Selection Strategies

In the previous experiments, all algorithms exhaustively perform lookups in the second layer. However, as we will see, the lookup selection policy plays an important role in the performance. Thus, we now look at better lookup strategies.

First, we investigate how query processing costs are distributed. Table 6 shows the average time overhead of each part of the query processing cost for 5000 access budget and 3000 lookups. Recall that query processing includes the online greedy depth selection, the radix sort, the scan for aggregating scores and lookup pruning, selection by sampling, lookups into the second layer, and another selection to get the c candidates. From Table 6, it is obvious that lookups are a large part of the total cost, as they involve decompression and access to many random blocks.

Parts of Query Processing	time (in ms)
Online greedy depth selection	0.0070
Radix-sort	0.1539
Scan for merge/aggregate/filter	0.1049
Selection	0.0311
Second-layer lookups	0.5394
Second selection	0.0275
Total	0.8638

Table 6: Query processing time breakdown.

Instead of exhaustively performing all lookups, we keep only a certain number m of candidates for lookups, based on partial BM25 score. To do this, we perform a randomized approximate selection as described earlier. We evaluate the quality and the speed of this lookup strategy for access budgets of 2000 and 5000 and a 0.5 pairwise space budget on top of the single-term structures of each access budget. Figure 3 presents the Overlap@$(500, 10)$ and the average query processing time for several configurations of m lookups, under both access budgets. More specifically, we allow lookups of $\{500, 1k, 2k, 3k, 5k\}$ candidates for access budget 5000, and $\{500, 750, 1k, 1.5k, 2k\}$ for access budget 2000 (from left to right). We see that better performance can be achieved with the proposed lookup method at the cost of some quality loss, and thus there is a trade-off between quality and time.

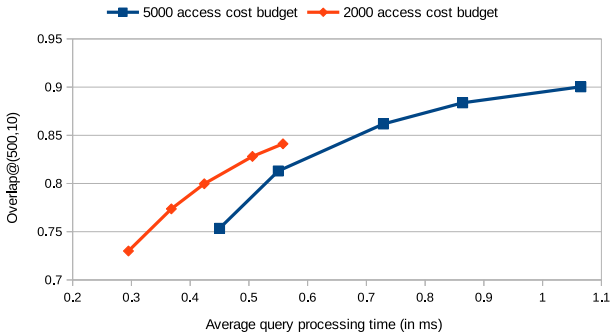

Figure 3: Effectiveness/Efficiency evaluation of our lookup strategy on Million09 queries.

Next, we compare the speed of our approach with other recently published methods. While top-10 query processing can be done in a few milliseconds [19], selecting the top 500 is much more expensive for most methods. Table 7 presents running times for different top-500 candidate generation methods. We implemented all methods in C++ and ran them on the ClueWeb09B data set. For safe methods, we pick BMW-LB, the fastest disjunctive method to our knowledge from [19], and BMA, the fastest conjunctive method from [21]. For unsafe methods, we implemented the best approach from [52], which is the tree-based Priority with pruning, and the best method from [43], BMW-CS, which uses 10% of the index as first layer. Our method with 5k-3k setup is much faster, since it only evaluates at most 5k postings, which means that less than 5k documents for each query are evaluated, while other methods usually evaluate many more documents. Both our online and offline posting selection mechanism contributes significantly to its good performance. BMA has the second fastest speed, since it is a conjunctive algorithm that only evaluates a limited number of documents. BMW-CS performs BMW on the first layer, which is the top 10 percent of the full index based on impact score. BMW-CS is different from our approach as we use pairwise structures in addition to singles in the first layer, and they do not have a fixed access budget per query or the greedy depth selection techniques. Their speed is much slower than ours, while still faster than BMW-LB. Priority is slower than BMW-LB, for top-500. For space overhead, BMA and BMW-CS both use less than 5% of the index size to store the Max-Block index. The two tiers in BMW-CS are disjoint, so the first tier takes no extra space. BMW-LB takes about 25% to cache the LB index, while priority takes no extra space. Our method with 5k-3k setup takes 59.9% extra space. Note that commercial search engines are often willing to accept significant space overheads for relatively small improvements in speed[42].

We now evaluate the effectiveness of our methods. Table 8 shows the Overlap@$(500, 10)$ for all the methods, and for

Method	BMW-LB	BMA	BMW-CS	Priority	5k-3k
Paper	[19]	[21]	[43]	[52]	Our
Time (ms)	26.74	15.12	19.31	51.24	0.863
Space	$\sim 25\%$	<5%	<5%	0%	59.9%

Table 7: Running times and space overhead of different approaches for top-500 documents retrieval.

our approach with three different setups. BMW-LB has the best result, and BMA is almost as good, but slightly worse. This means that conjunctive query processing is almost as good as disjunctive in terms of quality, which was also shown in [5]. In Table 9, we rerank the results of all the methods according to the complex ranker. BMW-LB+CF is much better than BMW-LB, which means that the complex ranker performs well on identifying more relevant results. The 5k-3k setup achieves consistently the best quality among our methods at all cutoff levels, and it's also slightly better than the other two unsafe methods for both Overlap and NDCG. Overall, we see that under NDCG, our approach can very quickly, in less than a millisecond, identify results that are almost as good as a safe disjunctive approach (BMW-LB).

BMW-LB	BMA	BMW-CS	Priority	5k-3k	2k-2k	2k-500
0.985	0.934	0.879	0.826	0.881	0.841	0.729

Table 8: Overlap@(500,10) for different methods.

Method / cutoff	1	5	10	20	100	200
BMW-LB+CF	0.267	0.253	0.259	0.252	0.275	0.292
BMA+CF	0.261	0.253	0.253	0.247	0.270	0.290
BMW-CS+CF	0.249	0.241	0.236	0.222	0.245	0.270
Priority+CF	0.261	0.242	0.239	0.243	0.252	0.269
BMW-LB	0.143	0.162	0.161	0.174	0.225	0.256
5k-3k+CF	0.266	0.256	0.256	0.246	0.269	0.289
2k-2k+CF	0.266	0.249	0.245	0.238	0.260	0.284
2k-5k+CF	0.254	0.250	0.241	0.228	0.242	0.261

Table 9: NDCG at various cutoff levels on Million09.

5.4 Varying Query Length and Candidates

Next, we test the impact of query length on speed and quality, using various configurations of our methods. Table 10 presents the average query processing time (ms) when varying the query length, whereas Table 11 reports the corresponding Overlap@$(500, 10)$ for each configuration. As query length increases, the query processing time also increases, since more lookups into more structures are performed. On the other hand, quality decreases, because top results in longer queries tend to be deeper inside the impact-sorted list (a basic fact in top-k query processing shown in Fagin's theoretical analysis of his algorithms [23]).

Setup	2	3	4	5	≥ 6	avg
2k-500	0.178	0.297	0.477	0.732	0.965	0.295
2k-2k	0.321	0.573	0.914	1.349	1.809	0.558
5k-3k	0.531	0.929	1.461	2.115	2.878	0.863

Table 10: Efficiency when varying query length on the Million09 queries with various setups, measured in ms.

Table 12 shows the impact of the number of candidates returned by our candidate generation algorithm (cutoff c) on Overlap@$(c, 10)$ for the 5000-3000 setup. As the cutoff c increases, the quality increases, untill flattening around 500. This justifies our choice of c = 500 throughout the paper.

Setup	2	3	4	5	≥ 6
2k-500	0.803	0.702	0.632	0.589	0.612
2k-2k	0.885	0.832	0.737	0.726	0.729
5k-3k	0.908	0.883	0.824	0.835	0.81

Table 11: Effectiveness in Overlap@$(500, 10)$ when we vary query length on Million09 queries in various setups.

c	100	200	300	400	500	800	1200
Overlap@$(c, 10)$	0.539	0.696	0.791	0.856	0.881	0.883	0.885

Table 12: Effectiveness when we vary candidates c.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a fast first-phase candidate generation approach for cascading ranking architectures. Our framework builds an auxiliary layer of index structures (single-term and pairwise structures) based on models for query term frequency and posting quality, which is then selectively accessed at query time based on a cost budget and using early termination techniques. The experimental evaluation shows that the proposed framework can find candidates about an order of magnitude faster than conjunctive or disjunctive top-k computations, with little loss in quality.

Future work includes the addition of specialized structures for phrases and proximity into our framework. We also expect some improvements from further optimization of the query processor and lookup mechanism, e.g., by adding bit vectors, or Bloom filters as in [4], or by using more complex rules to decide which lookups to perform (possibly based on ideas similar to [52]).

Acknowledgement

This research was supported by NSF Grant IIS-1117829 "Efficient Query Processing in Large Search Engines", and by a grant from Google.

7. REFERENCES

[1] D. Agarwal and M. Gurevich. Fast top-k retrieval for model based recommendation. In *Proc. of the Fifth Int. Conf. on Web Search and Data Mining*, pages 483–492, 2012.

[2] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. of the 29th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2006.

[3] N. Asadi. *Multi-Stage Search Architectures for Streaming Documents*. PhD thesis, University of Maryland, 2013.

[4] N. Asadi and J. Lin. Fast candidate generation for two-phase document ranking: postings list intersection with bloom filters. In *Proc. of the 21st ACM Conf. Information and Knowledge Management*, 2012.

[5] N. Asadi and J. Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *Proc. of the 36th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2013.

[6] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[7] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-Top-K: Index-access optimized top-k query processing. In *Proc. of the 32th Int. Conf. on Very Large Data Bases*, 2006.

[8] R. Blanco and A. Barreiro. Probabilistic static pruning of inverted files. *ACM Transactions on Information Systems*, 28(1), Jan. 2010.

[9] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[10] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of the 12th ACM Conf. on Information and Knowledge Management*, 2003.

[11] A. Broschart and R. Schenkel. High-performance processing of text queries with tunable pruned term and term pair indexes. *ACM Trans. Inf. Syst.*, 30(1):5, 2012.

[12] S. Büttcher and C. L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *Proc. of the 15th ACM Conf. Information and Knowledge Management*, 2006.

[13] 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. In *Proc. of the Third Int. Conf. on Web Search and Data Mining*, 2010.

[14] K. Chakrabarti, S. Chaudhuri, and V. Ganti. Interval-based pruning for top-k processing over compressed lists. In *Proc. of the 27th Int. Conf. on Data Engineering*, 2011.

[15] S. Chaudhuri, K. W. Church, A. C. König, and L. Sui. Heavy-tailed distributions and multi-keyword queries. In *Proc. of the 30th Annual Int. ACM SIGIR Conf*, 2007.

[16] G. V. Cormack, M. D. Smucker, and C. L. A. Clarke. Efficient and effective spam filtering and re-ranking for large web datasets. *Inf. Retr.*, 14(5):441–465, 2011.

[17] V. Dang, M. Bendersky, and W. B. Croft. Two-stage learning to rank for information retrieval. In *Proc. of the 35th European Conf. on Information Retrieval*, 2013.

[18] V. Dang and W. B. Croft. Query reformulation using anchor text. In *Proc. of the Third Int. Conf. on Web Search and Data Mining*, pages 41–50, 2010.

[19] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. A candidate filtering mechanism for fast top-k query processing on modern cpus. In *Proc. of the 36th Int. ACM SIGIR Conf*, 2013.

[20] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proc. of the Sixth Int. Conf. on Web Search and Data Mining*, 2013.

[21] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. of the 34th Int. ACM SIGIR Conf*, 2011.

[22] P. Donmez, K. M. Svore, and C. J. C. Burges. On the local optimality of lambdarank. In *Proc. of the 32th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2009.

[23] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31:2002, 2002.

[24] R. Fagin, D. Carmel, D. Cohen, E. Farchi, M. Herscovici, Y. Maarek, and A. Soffer. Static index pruning for information retrieval systems. In *Proc. of the 24th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2001.

[25] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, 2006.

[26] M. Fontoura, M. Gurevich, V. Josifovski, and S. Vassilvitskii. Efficiently encoding term co-occurrences in inverted indexes. In *Proc. of the 20th ACM Conf. Information and Knowledge Management*, 2011.

[27] S. Goel, J. Langford, and A. L. Strehl. Predictive indexing for fast search. In *Proc. of the 22nd Annual Conf. on Neural Information Processing Systems, Vancouver*, 2008.

[28] S. Hwang and K. C. Chang. Optimizing top-k queries for middleware access: A unified cost-based approach. *ACM Trans. Database Syst.*, 32(1):5, 2007.

[29] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[30] R. Kumar, K. Punera, T. Suel, and S. Vassilvitskii. Top-k aggregation using intersections of ranked inputs. In *Proc. of the Second Int. Conf. on Web Search and Data Mining*, 2009.

[31] H. Li. *Learning to Rank for Information Retrieval and Natural Language Processing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2011.

[32] T. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[33] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *Proc. of the 29th Int. Conf. on Very Large Data Bases*, 2003.

[34] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. *Proc. of the 15th Int. Conf. on World Wide Web*, 9(4), 2006.

[35] C. Macdonald, R. L. T. Santos, and I. Ounis. The whens and hows of learning to rank for web search. *Inf. Retr.*, 16(5):584–628, 2013.

[36] D. Metzler. Automatic feature selection in the markov random field model for information retrieval. In *Proc. of the 16th ACM Conf. Information and Knowledge Management*, 2007.

[37] A. Ntoulas and J. Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proc. of the 30th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2007.

[38] R. Ozcan, I. S. Altingövde, B. B. Cambazoglu, F. P. Junqueira, and Ö. Ulusoy. A five-level static cache architecture for web search engines. *Inf. Process. Manage.*, 48(5):828–840, 2012.

[39] M. Persin, J. Zobel, and R. Sacks-davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47:749–764, 1996.

[40] T. Qin, T. Liu, J. Xu, and H. Li. LETOR: A benchmark collection for research on learning to rank for information retrieval. *Inf. Retr.*, 13(4):346–374, 2010.

[41] K. Risvik, Y. Aasheim, and M. Lidal. Multi-tier architecture for web search engines. In *1st LA Web Congress*, 2003.

[42] K. M. Risvik, T. M. Chilimbi, H. Tan, K. Kalyanaraman, and C. Anderson. Maguro, a system for indexing and searching over very large text collections. In *Proc. of the Sixth Int. Conf. on Web Search and Data Mining*, 2013.

[43] C. Rossi, E. S. de Moura, A. L. Carvalho, and A. S. da Silva. Fast document-at-a-time query processing using two-tier indexes. In *Proc. of the 36th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2013.

[44] R. Schenkel, A. Broschart, S. Hwang, M. Theobald, and G. Weikum. Efficient text proximity search. In *Proc. of the 14th Int. Symposium on String Processing and Information Retrieval*, 2007.

[45] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of the 25th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2002.

[46] G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. A. Baeza-Yates. Resin: a combination of results caching and index pruning for high-performance web search engines. In *Proc. of the 31th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2008.

[47] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *Proc. of the 30th Annual Int. ACM SIGIR Conf*, 2007.

[48] T. Strohman, H. R. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *Proc. of the 28th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2005.

[49] N. Tonellotto, C. Macdonald, and I. Ounis. Efficient and effective retrieval using selective pruning. In *Proc. of the Sixth Int. Conf. on Web Search and Data Mining*, 2013.

[50] H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Inf. Processing and Management*, 31(6):831–850, 1995.

[51] L. Wang, J. J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proc. of the 34th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2011.

[52] H. Wu and H. Fang. Document prioritization for scalable query processing. In *Proc. of the 23rd ACM Conf. Information and Knowledge Management*, 2014.

[53] Q. Wu, C. J. C. Burges, K. M. Svore, and J. Gao. Adapting boosting for information retrieval measures. *Inf. Retr.*, 13(3):254–270, 2010.

[54] H. Yan, S. Shi, F. Zhang, T. Suel, and J. Wen. Efficient term proximity search with term-pair indexes. In *Proc. of the 19th ACM Conf. Information and Knowledge Management*, 2010.

[55] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. of the 17th Int. Conf. on World Wide Web*, 2008.

[56] M. Zhu, S. Shi, M. Li, and J. Wen. Effective top-k computation in retrieving structured documents with term-proximity support. In *Proc. of the 16th ACM Conf. Information and Knowledge Management*, 2007.

[57] M. Zhu, S. Shi, N. Yu, and J. Wen. Can phrase indexing help to process non-phrase queries? In *Proc. of the 17th ACM Conf. Information and Knowledge Management*, 2008.

[58] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surveys*, 38(2), 2006.

[59] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. of the 22th Int. Conf. on Data Engineering*, 2006.